Bachelor's Thesis

Bachelor's Degree in
Industrial Technology Engineering

Development and analysis of a NumPy-based DIC application for strain calculation

REPORT

Author: Marc Guillem Zamora Agustí
Director: Miquel Ferrer Ballester

Call: June 2018

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Summary

Digital Image Correlation (DIC) is a full-field, non-contact optical technique that tracks 2-D or 3-D images in order to establish a one-to-one correspondence to obtain accurate deformation data. This technique is being used more and more due to the technological improvements in computation and the rapid development of high resolution cameras.

This project aims to explain, first of all, an efficient algorithm to perform a DIC analysis, mostly based on the compilation of procedures done for Ncorr, a DIC application made by Justin Blaber as a completion of his master's degree. The algorithm uses the inverse compositional Gauss-Newton method (IC-GN) and interpolates the images by means of B-splines in a sophisticated way. An attempt has been made to describe the mathematical framework deeply enough to serve as the basis for developing an application.

A parallel objective of the project is, therefore, developing the application. The software is written in Python, but using mainly the scientific library NumPy, along with other C-based packages (SciPy, Cython, Matplotlib and PyInstaller) which allow, among other features, to manipulate image files and generate plots. Moreover, the vectorization technique, which has been necessary to speed up the code to a reasonable level, is outlined.

Finally, a brief set of tests are carried out to verify the accuracy and the reliability of the developed application, comparing the results with those obtained using the commercial package GOM Correlate. In the light of these results, it is clear that the application works appropriately, although further modifications need to be applied to improve the effectiveness of showing the strain data. Furthermore, another purpose is that the project goes ahead, since the application is quite basic, but there are still some programming techniques and other DIC improvements that could be tested and implemented. For this reason, a set of next steps have been listed at the end of this report.

Table of contents

1. Glossary

DFT	Discrete Fourier Transform
DIC	Digital Image Correlation
FA-GN	Forward Additive Gauss-Newton
FFT	Fast Fourier Transform
GN	Gauss-Newton
IC-GN	Inverse Compositional Gauss-Newton
RG	Reliability Guided
ROI	Region Of Interest
SIFT	Scale-Invariant Feature Transform
(Z)NCC	(Zero-mean) Normalized Cross-Correlation
ZNSSD	Zero-mean Normalized Sum of Squared Difference

2. Introduction

Digital Image Correlation (DIC) is a non-contact, optical technique that aims to track 2-D or 3-D images to obtain accurate displacement and deformation data. The most common application of DIC is to compute full field displacements and strains of objects undergoing physical deformations due to acting forces. In recent years different techniques have been developed both to improve the reliability and accuracy of the algorithm and its performance.

There is an evident growing interest in this technique, due the continuous improvements in computing power and to its simple experimental setup. DIC takes advantage of the speckle pattern of the specimen, which can be either intrinsic or artificial, being able to correlate the images by means of comparing the pixel intensity within small areas, the so-called subsets. Each material point in the reference configuration is the centre of a subset, and then the subset is deformed until it best matches the final configuration, finding the corresponding current material point. The reference material points, which correspond to integer pixel locations, are usually separated a certain number of pixels, so that the computation time is reduced.

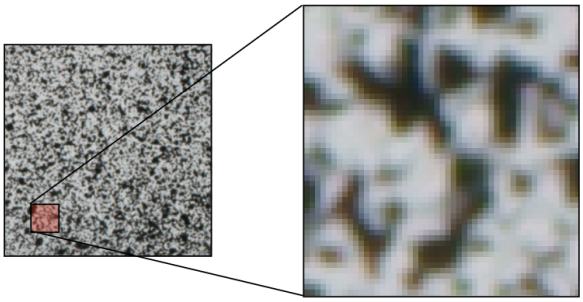


Figure 2.1 – Example of the speckle surface of a specimen.

Once this one-to-one correspondence has been found, that is, the relative displacements between images, further parameters can be calculated, such as the strains. Given that the main purpose of DIC is to find the aforementioned correspondence, the calculation of the strains is explained separately in section 4, which also includes the algorithm to find the full field displacements. Section 3 outlines the algorithm to track a single material point.

2.1. Objectives of the project and methodology

The first main objective of the project is to describe in detail a set of procedures necessary to carry out a 2-D DIC analysis. There are many different approaches to it, but the chosen techniques [2,3] are based on Ncorr, a software developed by Justin Blaber as a completion of his master's degree at Georgia Institute of Technology.

Although there are several commercial packages that allow to develop DIC analyses, such as VIC-2D or GOM Correlate, another goal is to develop an application using scientific libraries for Python, such as NumPy and SciPy, thus laying the groundwork for testing different techniques that can be found in the literature. The code has been written using Sublime Text 3 and Anaconda, and the following 64-bit Python libraries (which have been verified to be compatible) that must be installed using *pip* with following *wheel* files:

- matplotlib-2.1.2-cp36-cp36m-win_amd64.whl
- numba-0.37.0-cp36-cp36m-win_amd64.whl
- numpy-1.14.1+mkl-cp36-cp36m-win_amd64.whl
- opencv_python-3.4.1-cp36-cp36m-win_amd64.whl
- scipy-1.0.0-cp36-cp36m-win_amd64.whl

The third main objective is to verify that the application works correctly, by testing images the results of which are known, and also by contrasting the results with those obtained using the commercial package GOM Correlate.

Lastly, an objective shared by the previous ones is to open a path for improvements, so that the project is able to go ahead.

2.2. Scope of the project

This project does not explore any DIC techniques that cannot be found in the literature. In this regard, it is a compilation of a set of efficient techniques that have been already verified to yield good results, and which are explained throughout the report linking the different ideas to form a complete analysis scheme.

As for the programming part, the structure of the code and its functions are presented, as well as highlighting the importance of vectorizing the functions to achieve reasonable levels of speed. On the other hand, it is based only on Python and the aforementioned libraries, thus some of the described procedures cannot be extrapolated to other languages.

Finally, it is worth noting that the nomenclature and the definition of some objects may not correspond to those found in the literature, except for those that are almost always represented equally, such as the image coordinates (x, y) and the displacements (u, v).

3. DIC algorithm

3.1. Subset transformation

Subsets can be defined in different ways, being their common properties the size and the shape. In this project, square subsets have been chosen, and their size (number of pixels of the square side) is a parameter that must be defined prior to the DIC analysis. Arbitrarily, the centre (x_c, y_c) of a subset is positioned in such a way that the points x_i of x axis form the sequence $\{x_c - \lfloor \frac{size}{2} \rfloor \ , \dots , x_c, \dots , x_c + \lceil \frac{size}{2} \rceil - 1\}$, and similarly for axis y. Figure 3.1 shows an example of a subset configuration, taking $size = 12 \ px$.

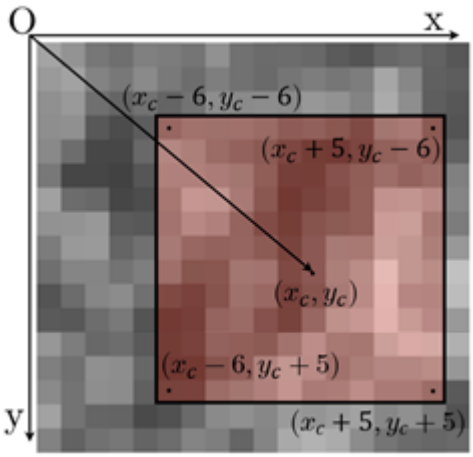


Figure 3.1 – Example of a subset configuration.

The deformation within each subset S is supposed uniform, thus choosing a larger or smaller size can lead, respectively, to unacceptable estimation errors or to a wrong matching between images (because too small subsets can be confused due to their similarity). This deformation is given by the linear, first order transformation in Eq. 3.1 [2,3,12]. A second order transformation may be used instead if the deformation between images is more complex [7], although it increases the computational costs significantly, and might be solved by choosing smaller subsets or by taking more pictures and performing intermediate analyses.

$$\tilde{x}_{w_i} = x_{t_i} + u + \frac{\partial u}{\partial x}\left(x_{t_i} - x_{t_c}\right) + \frac{\partial u}{\partial y}\left(y_{t_j} - y_{t_c}\right)$$

$$\tilde{y}_{w_j} = y_{t_j} + v + \frac{\partial v}{\partial x}\left(x_{t_i} - x_{t_c}\right) + \frac{\partial v}{\partial y}\left(y_{t_j} - y_{t_c}\right) \qquad (i,j) \in S \qquad \text{(Eq. 3.1)}$$

In Eq. 3.1, ~ refers to the transformed point; u, v are the horizontal and vertical displacements, respectively; the subscript c means *centre*, and t, w refer to the image where the points will be evaluated to obtain grayscale data. It is worth noting that when $t \neq w$ the transformation will occur between the reference and the current image, whereas the case where $t = w$ (evaluating a deformed subset using grayscale data from the reference image) will be needed for the type of algorithm developed, as will be explained throughout section 3.

Eq. 3.1 can also be written, for later clarification purposes, as follows:

$$\begin{Bmatrix} \tilde{x}_{w_i} \\ \tilde{y}_{w_j} \\ 1 \end{Bmatrix} = \mathbf{O_v} + W(\mathbf{\Delta X_{ij}}; \mathbf{P_{tw}}) = \begin{Bmatrix} x_{t_c} \\ y_{t_c} \\ 0 \end{Bmatrix} + \begin{bmatrix} 1 + \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & u \\ \frac{\partial v}{\partial x} & 1 + \frac{\partial v}{\partial y} & v \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} \Delta x_{t_i} \\ \Delta y_{t_j} \\ 1 \end{Bmatrix} \qquad \text{(Eq. 3.2)}$$

$$\Delta x_{t_i} = x_{t_i} - x_{t_c} \quad ; \quad \Delta y_{t_j} = y_{t_j} - y_{t_c}$$

In Eq. 3.2, $W(\mathbf{\Delta X_{ij}}; \mathbf{P_{tw}})$ is a warping function, which takes points from t to w according to the deformation vector $\mathbf{P_{tw}} = \left\{u \quad v \quad \frac{\partial u}{\partial x} \quad \frac{\partial u}{\partial y} \quad \frac{\partial v}{\partial x} \quad \frac{\partial v}{\partial y}\right\}^T$. Finding this deformation vector is the main goal in order to stablish a correspondence between the reference state and the current state, and from these results further calculations may be performed to obtain strain data.

The most straightforward and intuitive approach seems to be finding $\mathbf{P_{tw}}$ by means of some kind of procedure involving correlation functions, between the undeformed subset of the reference state and the resulting deformed subset corresponding to the current state. Notwithstanding, the subset of the reference state is allowed to deform within the same state. Expressing the aforementioned idea in a generic and more appropriate way for the algorithm to be developed, the core objective is to find the optimal $\mathbf{P_{tw}}$, namely $\mathbf{P^*_{tw}}$, for each subset $\tilde{S}_t$

which best matches the deformed subset $\tilde{S}_w$ when the first is evaluated at the reference state, and the second at the current state, and also $\tilde{S}_t = S_t$, where S_t is the undeformed subset, chosen before the start of the DIC algorithm. That is to say, for a given $\mathbf{P_{tw}}$, S_t does not need to deform to $\tilde{S}_t$ ($\mathbf{P_{tt}} = \mathbf{0}$) to achieve optimality in the way it has been defined. In this situation, the solution to the intuitive approach has been found, but it is worth noting that the aforementioned idea does not constrain intermediate calculations from deforming the reference subset within the reference configuration.

3.2. Metrics

The values that make the vector $\mathbf{P}^*_{\mathbf{tw}}$ optimal are those that transform the original subset S_t to $\tilde{S}_w$ in such a way that the correlation between them is maximum, in the sense of the grayscale level of the pixels, which is the only information gathered from the object. Therefore, it seems reasonable to consider some kind of coefficient to quantify the relationship between the undeformed and the deformed subset. Two measurement coefficients will be used in the algorithm to yield these results: the zero-mean normalized cross-correlation (C_{ZNCC}) and the zero-mean normalized sum of squared difference (C_{ZNSSD}) [2,3,12,14]. The first one indicates a good match when it is close to 1, whilst the second one indicates a good match for values close to 0. Being $G(x_s, y_s) \in [0,1]$ the grayscale value at a point (x, y) and evaluating it at the state s, they are defined as follows:

$$C_{ZNCC} = \frac{\sum_{(i,j)\in S}[G(\tilde{x}_{t_i}, \tilde{y}_{t_j}) - \bar{G}(\tilde{S}_t)][G\left(\tilde{x}_{w_i}, \tilde{y}_{w_j}\right) - \bar{G}(\tilde{S}_w)]}{\sqrt{\sum_{(i,j)\in S}\left[G\left(\tilde{x}_{t_i}, \tilde{y}_{t_j}\right) - \bar{G}\left(\tilde{S}_t\right)\right]^2 \sum_{(i,j)\in S}\left[G\left(\tilde{x}_{w_i}, \tilde{y}_{w_j}\right) - \bar{G}\left(\tilde{S}_w\right)\right]^2}} \quad \text{(Eq. 3.3)}$$

$$C_{ZNSSD} = \sum_{(i,j)\in S}\left[\frac{G\left(\tilde{x}_{t_i}, \tilde{y}_{t_j}\right) - \bar{G}\left(\tilde{S}_t\right)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(\tilde{x}_{t_i}, \tilde{y}_{t_j}\right) - \bar{G}\left(\tilde{S}_t\right)\right]^2}} - \frac{G\left(\tilde{x}_{w_i}, \tilde{y}_{w_j}\right) - \bar{G}\left(\tilde{S}_w\right)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(\tilde{x}_{w_i}, \tilde{y}_{w_j}\right) - \bar{G}\left(\tilde{S}_w\right)\right]^2}}\right]^2 \quad \text{(Eq. 3.4)}$$

In both Eq. 3.3 and Eq. 3.4, $\bar{G}(S)$ is the mean of the grayscale values of the subset S, which is clearly $\sum_{(i,j)\in S}\frac{G(x_i,y_j)}{N}$, being N the number of points of the subset. Besides, it is worth mentioning that these equations are defined in a generic way, so that they serve both for transformations between $\tilde{S}_t \neq S_t$ and $\tilde{S}_w$ and between $\tilde{S}_t = S_t$ and $\tilde{S}_w$, with $\mathbf{P_{tt}} = \mathbf{0}$ in this last case.

The main advantage of C_{ZNCC} and C_{ZNSSD} over other correlation criteria is that they are both insensitive to offset and scale changes of the intensity of the current state. Furthermore, they are related by the equation $C_{ZNSSD} = 2(1 - C_{ZNCC})$ [14], but the reason for using one or the other is based on their adequacy in certain calculations. Basically, the Taylor series of C_{ZNSSD} will allow to establish an iteration scheme to find $\mathbf{P}^*_{\mathbf{tw}}$, whereas C_{ZNCC} will yield an initial guess.

3.3. Gauss-Newton algorithm

The Gauss-Newton algorithm [2,3,12] is used to find the roots of a univariate or multivariate function, or its derivatives, by means of an iterative scheme. In this case, the minimum value of C_{ZNSSD} for a particular reference subset is desired, and will be given by $\mathbf{P}^*_{\mathbf{tw}}$. The next expression is the first order Taylor series expansion of C_{ZNSSD} around $\mathbf{P}_{\mathbf{0}}^{(n)}$, which represents the guess obtained from the previous iteration $n-1$ (or the initial guess in the first iteration).

$$C_{ZNSSD}\left(\mathbf{P}_{\mathbf{0}}^{(n)} + \mathbf{P}^{(n)}\right) \approx C_{ZNSSD}\left(\mathbf{P}_{\mathbf{0}}^{(n)}\right) + \nabla C_{ZNSSD}\left(\mathbf{P}_{\mathbf{0}}^{(n)}\right)^T \mathbf{P}^{(n)} \qquad \text{(Eq. 3.5)}$$

Taking the derivative with respect to $\mathbf{P}^{(n)}$ in Eq. 3.5 and making it equal to zero, since the minimum is desired, yields:

$$\nabla\nabla C_{ZNSSD}\left(\mathbf{P}_{\mathbf{0}}^{(n)}\right)\mathbf{P}^{(n)} = -\nabla C_{ZNSSD}\left(\mathbf{P}_{\mathbf{0}}^{(n)}\right) \qquad \text{(Eq. 3.6)}$$

This is, in fact, the Newton-Raphson method (the next guess $\mathbf{P}_{\mathbf{0}}^{(n+1)}$ will be $\mathbf{P}_{\mathbf{0}}^{(n)} + \mathbf{P}^{(n)}$, or an equivalent). From simplifying the Hessian matrix $\nabla\nabla C_{ZNSSD}$, it becomes the Gauss-Newton algorithm, as explained in 3.3.1.

3.3.1. Forward additive Gauss-Newton method (FA-GN)

Although this method will not be implemented, it is briefly presented here since it offers a good introduction to the chosen one and because it is analogous to the process that would be applied intuitively. In fact, it is the standard application of the GN iterative scheme.

In this method, the reference subset is not allowed to deform within the reference configuration, that is, $\mathbf{P_{tt}} = \mathbf{0}$, and the deformation vector $\mathbf{P}^{(n)}$ becomes $\mathbf{P}^{(n)}_{\mathbf{tw}}$. Redefining Eq. 3.4 in terms of $\mathbf{P}^{(n)}_{\mathbf{0}} + \mathbf{P}^{(n)}_{\mathbf{tw}}$ and the notation of Eq. 3.2, it becomes:

$$C_{ZNSSD}\left(\mathbf{P}^{(n)}_{\mathbf{0}} + \mathbf{P}^{(n)}_{\mathbf{tw}}\right) =$$

$$\sum_{(i,j)\in S}\left[\frac{\dfrac{G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)}{\sqrt{\sum_{(i,j)\in S}\left[G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)\right]^2}} - \dfrac{G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P}^{(n)}_{\mathbf{0}}+\mathbf{P}^{(n)}_{\mathbf{tw}}\right)\right)-\bar{G}\left(\tilde{S}_w\right)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P}^{(n)}_{\mathbf{0}}+\mathbf{P}^{(n)}_{\mathbf{tw}}\right)\right)-\bar{G}\left(\tilde{S}_w\right)\right]^2}}\right]^2 \quad \text{(Eq. 3.7)}$$

In order to simplify the calculations, the following assumptions are made, which are valid as long as the mean of the grayscale values of the current subset does not vary much for deformations around $\mathbf{P_0}$.

$$\left.\frac{d}{d\mathbf{P}^{(n)}_{\mathbf{tw}}}\right|_{\mathbf{P}^{(n)}_{\mathbf{0}}} \bar{G}\left(\tilde{S}_w\right) = 0 \quad \text{(Eq. 3.8)}$$

$$\left.\frac{d}{d\mathbf{P}^{(n)}_{\mathbf{tw}}}\right|_{\mathbf{P}^{(n)}_{\mathbf{0}}} \sqrt{\sum_{(i,j)\in S}\left[G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P}^{(n)}_{\mathbf{tw}}\right)\right)-\bar{G}\left(\tilde{S}_w\right)\right]^2} = 0 \quad \text{(Eq. 3.9)}$$

The third simplification arises from considering that the initial guess is close to the optimal solution, which is valid if the deformation is small enough so that NCC yields an appropriate displacement vector. This assumption, given by the expression in Eq. 3.10, converts the

Newton-Raphson algorithm into the Gauss-Newton method, which has better convergence characteristics, along with a simpler implementation.

$$\sum_{(i,j)\in S}\left[\frac{\dfrac{G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)}{\sqrt{\sum_{(i,j)\in S}\left[G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)\right]^2}}-\dfrac{G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P_0^{(n)}}\right)\right)-\bar{G}\left(\tilde{S}_w\right)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P_0^{(n)}}\right)\right)-\bar{G}\left(\tilde{S}_w\right)\right]^2}}\right]. \tag{Eq. 3.10}$$

$$\left[\frac{d^2}{d\left(\mathbf{P_{tw}^{(n)}}\right)^2}\bigg|_{\mathbf{P_0^{(n)}}} G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P_{tw}^{(n)}}\right)\right)\right]=0$$

Eq. 3.10 is also reasonable if the second derivatives of the grayscale values of the current subset evaluated at $\mathbf{P_0^{(n)}}$ are small enough, which is generally true, as the grayscale values are usually smooth. Nonetheless, this also depends on the interpolation method and degree, so choosing one or another will make this assumption more or less accurate, together with having a guess close to $\mathbf{P_{tw}^{*}}$.

The expressions of the gradient and the Hessian of C_{ZNSSD} can be found in [2], with which Eq. 3.6 is solved for $\mathbf{P}^{(n)}=\mathbf{P_{tw}^{(n)}}$ using Cholesky decomposition and forward-back substitution. The guess of the next iteration is $\mathbf{P_0^{(n+1)}}=\mathbf{P_{tw}^{(n)}}+\mathbf{P_0^{(n)}}$, and this is what gives the method its name, since this new guess is the previous guess plus the solution from the previous iteration, which is always between the reference subset within the reference configuration and the deformed subset within the current state. Figure 3.2 shows a graphic example of this method.

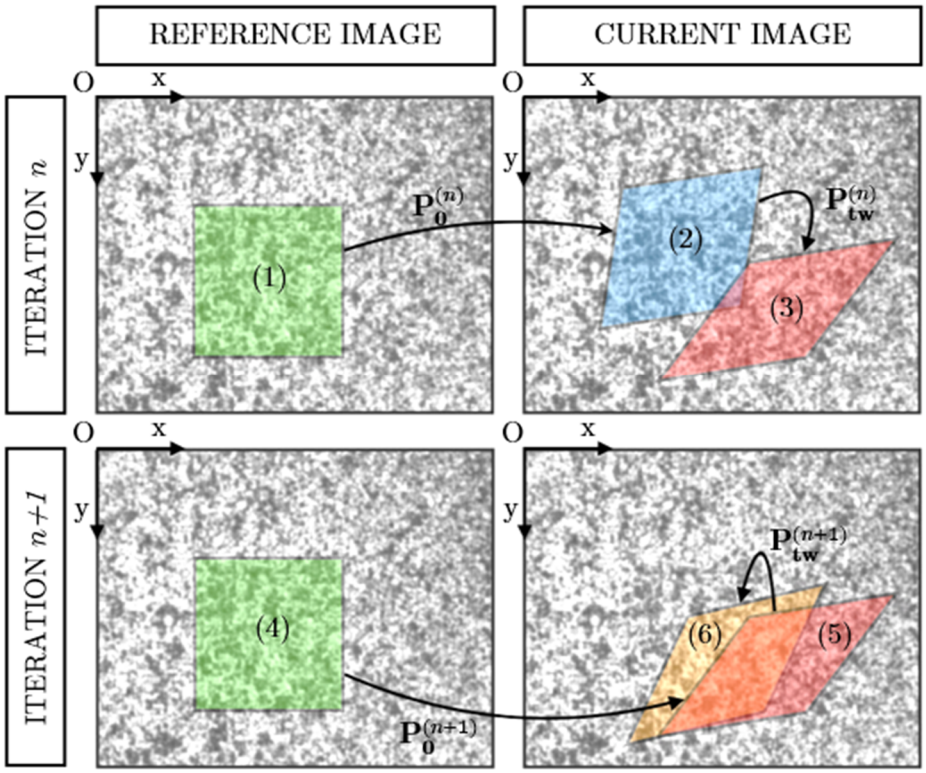


Figure 3.2 – Example of the FA-GN method. Same colour indicates same subset. The numbers indicate the progression. The deformations are exaggerated, for clarity purposes.

3.3.2. Inverse compositional Gauss-Newton method (IC-GN)

The core idea is common to both methods: to find, in every iteration, an intermediate solution that allows to calculate a closer solution to the optimal one. The main differences between this method and the FA-GN method is that here the reference subset is allowed to deform within the reference configuration, whereas $\mathbf{P}_{\mathbf{tw}}^{(n)}$ is set constant in every iteration, which is updated according to the results of the last iteration [2,3,9]. $\mathbf{P}_{\mathbf{tw}}^{(0)}$ is the initial guess found by means of NCC. The optimal deformation vector, as mentioned before, has $\mathbf{P}_{\mathbf{tt}} = 0$ associated. Therefore, $\mathbf{P}_{\mathbf{0}}^{(n)}$ from Eq. 3.6 becomes 0, and $\mathbf{P}^{(n)}$ becomes $\mathbf{P}_{\mathbf{tt}}^{(n)}$, so $C_{ZNSSD} = C_{ZNSSD}\left(\mathbf{P}_{\mathbf{tt}}^{(n)}\right)$. Under these considerations, Eq. 3.7 takes the following form.

$$C_{ZNSSD}\left(\mathbf{P}_{\mathbf{tt}}^{(n)}\right) =$$

$$\sum_{(i,j)\in S}\left[\frac{G\left(\mathbf{O}_{\boldsymbol{v}}+W\left(\boldsymbol{\Delta}\mathbf{X}_{\mathbf{ij}};\mathbf{P}_{\mathbf{tt}}^{(n)}\right)\right)-\bar{G}\left(\tilde{S}_t\right)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(\mathbf{O}_{\boldsymbol{v}}+W\left(\boldsymbol{\Delta}\mathbf{X}_{\mathbf{ij}};\mathbf{P}_{\mathbf{tt}}^{(n)}\right)\right)-\bar{G}\left(\tilde{S}_t\right)\right]^2}}-\frac{G\left(\mathbf{O}_{\boldsymbol{v}}+W\left(\boldsymbol{\Delta}\mathbf{X}_{\mathbf{ij}};\mathbf{P}_{\mathbf{tw}}^{(n)}\right)\right)-\bar{G}\left(\tilde{S}_w\right)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(\mathbf{O}_{\boldsymbol{v}}+W\left(\boldsymbol{\Delta}\mathbf{X}_{\mathbf{ij}};\mathbf{P}_{\mathbf{tw}}^{(n)}\right)\right)-\bar{G}\left(\tilde{S}_w\right)\right]^2}}\right]^2 \quad \text{(Eq. 3.11)}$$

As with FA-GN method, the equivalent assumptions given by Eq. 3.8, Eq. 3.9 and Eq. 3.10 are considered (see section 3.3.3). Once again, computing them will yield $\mathbf{P}_{\mathbf{tt}}^{(n)}$ by means of Cholesky decomposition and forward-back substitution.

From $\mathbf{P}_{\mathbf{tt}}^{(n)}$, there is a need to get closer to the optimal solution in terms of $\mathbf{P}_{\mathbf{tw}}$. First of all, the situation shown in Figure 3.3 is considered.

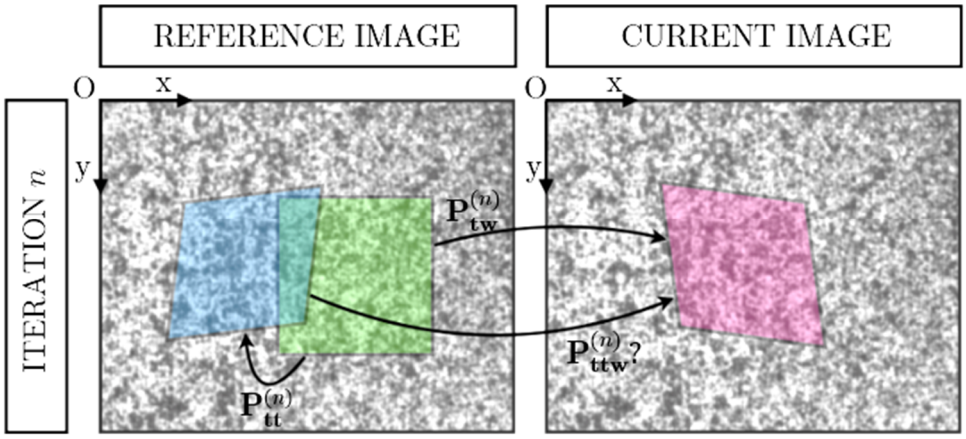


Figure 3.3 – Example of the IC-GN method which shows the desired $\boldsymbol{P}_{\boldsymbol{ttw}}^{(n)}$ vector.

In order to find $\mathbf{P}_{\mathbf{ttw}}^{(n)}$, by manipulating Eq. 3.2 the following results are obtained, where $M_A^{(n)}$ is the matrix that appears in the original equation, with the components of $\mathbf{P}_{\boldsymbol{A}}^{(n)}$.

$$\begin{Bmatrix} \tilde{x}_{t_i} - x_{t_c} \\ \tilde{y}_{t_j} - y_{t_c} \\ 1 \end{Bmatrix} = M_{tt}^{(n)} \begin{Bmatrix} \Delta x_{t_i} \\ \Delta y_{t_j} \\ 1 \end{Bmatrix} \quad ; \quad \begin{Bmatrix} \tilde{x}_{w_i} - x_{t_c} \\ \tilde{y}_{w_j} - y_{t_c} \\ 1 \end{Bmatrix} = M_{tw}^{(n)} \begin{Bmatrix} \Delta x_{t_i} \\ \Delta y_{t_j} \\ 1 \end{Bmatrix} \tag{Eq. 3.12}$$

$M_{ttw}^{(n)}$ can be obtained by operating the expressions in Eq. 3.12:

$$\begin{Bmatrix} \tilde{x}_{w_i} - x_{t_c} \\ \tilde{y}_{w_j} - y_{t_c} \\ 1 \end{Bmatrix} = M_{tw}^{(n)} \left[M_{tt}^{(n)} \right]^{-1} \begin{Bmatrix} \tilde{x}_{t_i} - x_{t_c} \\ \tilde{y}_{t_j} - y_{t_c} \\ 1 \end{Bmatrix} \approx M_{ttw}^{(n)} \begin{Bmatrix} \Delta \tilde{x}_{t_i} \\ \Delta \tilde{y}_{t_j} \\ 1 \end{Bmatrix} \tag{Eq. 3.13}$$

Then, from $M_{ttw}^{(n)}$, $\mathbf{P}_{\mathbf{ttw}}^{(n)}$ can be extracted. It is important to note that the approximation in Eq. 3.13 is valid as long as (u, v) from $\mathbf{P}_{\mathbf{tt}}^{(n)}$ are small enough. If this is true, then the centres of the reference subset and the deformed subset, both within the reference configuration, will be close enough to assume that $x_{t_c} = \tilde{x}_{t_c}$ and $y_{t_c} = \tilde{y}_{t_c}$. Therefore, $\mathbf{P}_{\mathbf{ttw}}^{(n)}$ will define a transformation between $\tilde{S}_t$ and $\tilde{S}_w$ which takes to the same subset as $\mathbf{P}_{\mathbf{tw}}^{(n)}$ from S_t. In fact, $\mathbf{P}_{\mathbf{ttw}}^{(n)}$ is the new $\mathbf{P}_{\mathbf{tw}}$ of the next iteration, $\mathbf{P}_{\mathbf{tw}}^{(n+1)} = \mathbf{P}_{\mathbf{ttw}}^{(n)}$. It is worth noting that the transformation described by a deformation vector can be applied to another initial subset close to the original one, and the final subset will be, consequently, close to the original current subset (regarding grayscale values), more or less depending on the magnitude of the deformations. Therefore, it follows that $\mathbf{P}_{\mathbf{ttw}}^{(n)}$ can be applied to S_t to form a current subset close to $\tilde{S}_w$ but closer than $\tilde{S}_w$ to the optimal solution. Clearly, in the next iterations $\mathbf{P}_{\mathbf{tt}}$ will be eventually smaller than in the previous one, and the iterative scheme can be stopped when the magnitude of this deformation vector is close to 0. Figure 3.4 shows a graphic example of the process.

Compared to the FA-GN method, the IC-GN algorithm is more complex to implement, but it has a superior advantage regarding computational cost: the Hessian, which is expensive to calculate, only needs to be computed once for all the iterations, given the fact that $\mathbf{P}_{\mathbf{0}}^{(n)}$ in Eq. 3.6 is set to 0 at the beginning of each iteration. This is not the case with the FA-GN method, where $\mathbf{P}_{\mathbf{0}}^{(n)}$ is new in each iteration. Moreover, despite the need of updating $\mathbf{P}_{\mathbf{tw}}^{(n)}$ with two 6×6 matrix operations in front of an addition, this kind of calculation is much faster than the Hessian one, which involves many arithmetic operations and it depends on the subset size.

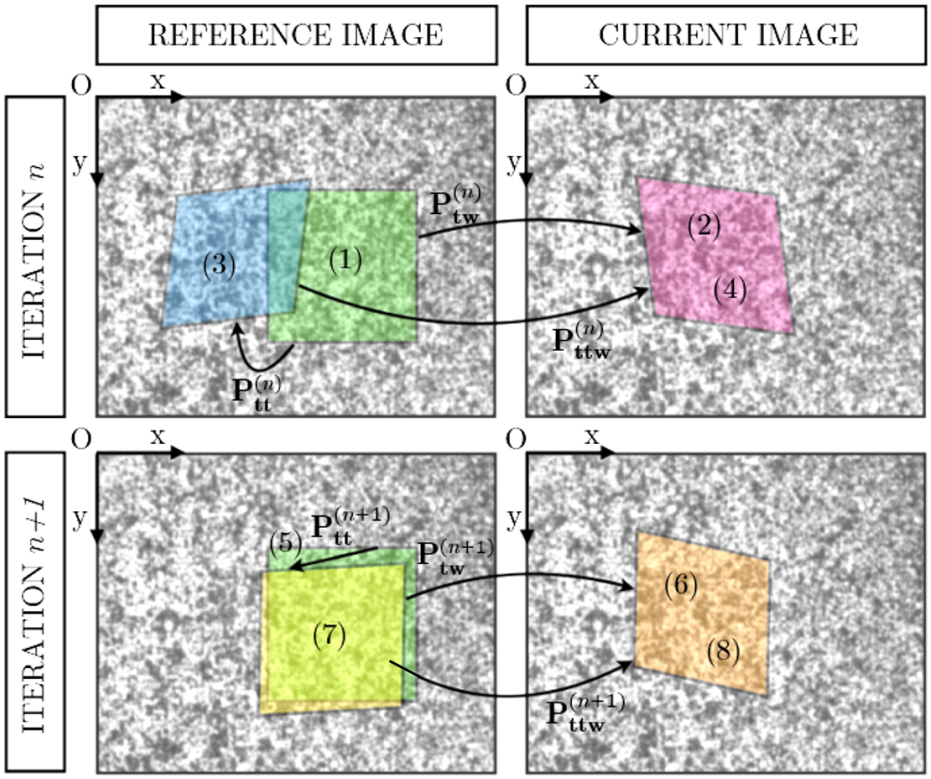


Figure 3.4 - Example of the IC-GN method. Same colour indicates same subset. The numbers indicate the progression. The deformations are exaggerated, for clarity purposes.

3.3.3. Gradient and Hessian of C_{ZNSSD} for IC-GN method

Firstly, the equivalent assumptions described by Eq. 3.8, Eq. 3.9 and Eq. 3.10 are presented below.

$$\left.\frac{d}{d\mathbf{P}_{\mathbf{tt}}^{(n)}}\right|_{\mathbf{P}_{\mathbf{tt}}^{(n)}=\mathbf{0}} \bar{G}\left(\tilde{S}_t\right) = 0 \qquad \text{(Eq. 3.14)}$$

$$\left.\frac{d}{d\mathbf{P}_{\mathbf{tt}}^{(n)}}\right|_{\mathbf{P}_{\mathbf{tt}}^{(n)}=\mathbf{0}} \sqrt{\sum_{(i,j)\in S}\left[G\left(\mathbf{O}_{\boldsymbol{v}} + W(\boldsymbol{\Delta}\mathbf{X}_{\mathbf{ij}}, \mathbf{P}_{\mathbf{tt}}^{(n)})\right) - \bar{G}\left(\tilde{S}_t\right)\right]^2} = 0 \qquad \text{(Eq. 3.15)}$$

$$\sum_{(i,j)\in S}\left[\frac{\dfrac{G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)}{\sqrt{\sum_{(i,j)\in S}\left[G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)\right]^2}}-\dfrac{G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P_{tw}^{(n)}}\right)\right)-\bar{G}\left(\tilde{S}_w\right)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P_{tw}^{(n)}}\right)\right)-\bar{G}\left(\tilde{S}_w\right)\right]^2}}\right]\cdot \tag{Eq. 3.16}$$

$$\left[\left.\frac{d^2}{d\left(\mathbf{P_{tt}^{(n)}}\right)^2}\right|_{\mathbf{P_{tt}^{(n)}}=\mathbf{0}} G\left(\mathbf{O_v}+W(\mathbf{\Delta X_{ij}},\mathbf{P_{tt}^{(n)}})\right)\right]=0$$

The expressions of the gradient and the Hessian of C_{ZNSSD} with respect to $\mathbf{P_{tt}^{(n)}}$ at $\mathbf{P_{tt}}=\mathbf{0}$, which already take into account the aforementioned assumptions, are presented below.

$$\nabla C_{ZNSSD}(\mathbf{0})\approx$$

$$\frac{2}{\sqrt{\sum_{(i,j)\in S}\left[G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)\right]^2}}\cdot$$

$$\sum_{(i,j)\in S}\left[\left[\frac{\dfrac{G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)}{\sqrt{\sum_{(i,j)\in S}\left[G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)\right]^2}}-\dfrac{G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P_{tw}^{(n)}}\right)\right)-\bar{G}\left(\tilde{S}_w\right)}{\sqrt{\sum_{(i,j)\in S}\left[G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P_{tw}^{(n)}}\right)\right)-\bar{G}\left(\tilde{S}_w\right)\right]^2}}\right]\cdot\left[\left.\frac{d}{d\left(\mathbf{P_{tt}^{(n)}}\right)}\right|_{\mathbf{P_{tt}^{(n)}}=\mathbf{0}} G\left(\mathbf{O_v}+W(\mathbf{\Delta X_{ij}},\mathbf{P_{tt}^{(n)}})\right)\right]\right] \tag{Eq. 3.17}$$

$$\nabla\nabla C_{ZNSSD}(\mathbf{0}) \approx$$

$$\frac{2}{\sum_{(i,j)\in S}\left[G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})-\bar{G}(S_t)\right]^2}\cdot$$

$$\sum_{(i,j)\in S}\left[\left[\left.\frac{d}{d\left(\mathbf{P_{tt}^{(n)}}\right)}\right|_{\mathbf{P_{tt}^{(n)}}=\mathbf{0}} G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}},\mathbf{P_{tt}^{(n)}}\right)\right)\right]\left[\left.\frac{d}{d\left(\mathbf{P_{tt}^{(n)}}\right)}\right|_{\mathbf{P_{tt}^{(n)}}=\mathbf{0}} G\left(\mathbf{O_v}+W(\mathbf{\Delta X_{ij}},\mathbf{P_{tt}^{(n)}})\right)\right]^T\right] \quad \text{(Eq. 3.18)}$$

The quantities $G(\mathbf{O_v}+\mathbf{\Delta X_{ij}})$ from Eq. 3.17 and Eq. 3.18 can be computed straightforwardly, as the points correspond to integer pixel locations within the reference configuration. On the other hand, the quantities $G\left(\mathbf{O_v}+W\left(\mathbf{\Delta X_{ij}};\mathbf{P_{tw}^{(n)}}\right)\right)$ and all the derivatives require some kind of interpolation. This will be covered in the next section.

3.3.4. Calculation of the gradient and the Hessian

In order to calculate the term $\left.\frac{d}{d\left(\mathbf{P_{tt}^{(n)}}\right)}\right|_{\mathbf{P_{tt}^{(n)}}=\mathbf{0}} G\left(\mathbf{O_v}+W(\mathbf{\Delta X_{ij}},\mathbf{P_{tt}^{(n)}})\right)$ from both the gradient and the Hessian of $C_{ZNSSD}(\mathbf{0})$, the chain rule is applied in the first place [2,3].

$$\left.\frac{d}{d\mathbf{P_{tt}^{(n)}}}\right|_{\mathbf{P_{tt}^{(n)}}=\mathbf{0}} G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)=\frac{\partial}{\partial\tilde{x}_{t_i}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)\frac{d\tilde{x}_{t_i}}{d\mathbf{P_{tt}^{(n)}}}+\frac{\partial}{\partial\tilde{y}_{t_j}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)\frac{d\tilde{y}_{t_j}}{d\mathbf{P_{tt}^{(n)}}} \quad \text{(Eq. 3.19)}$$

Secondly, from Eq. 3.19 and the general expression given by Eq. 3.2, the derivatives with respect to each element of the deformation vector $\mathbf{P_{tt}^{(n)}}=\left\{u \quad v \quad \frac{\partial u}{\partial x} \quad \frac{\partial u}{\partial y} \quad \frac{\partial v}{\partial x} \quad \frac{\partial v}{\partial y}\right\}^T$ take the following forms.

$$\frac{d}{du}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)=\frac{\partial}{\partial\tilde{x}_{t_i}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right) \quad \text{(Eq. 3.20)}$$

$$\frac{d}{dv}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)=\frac{\partial}{\partial\tilde{y}_{t_j}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right) \quad \text{(Eq. 3.21)}$$

$$\frac{d}{d\left(\frac{\partial u}{\partial x}\right)}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)=\frac{\partial}{\partial\tilde{x}_{t_i}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)\Delta x_{t_i} \quad \text{(Eq. 3.22)}$$

$$\frac{d}{d(\frac{\partial u}{\partial y})}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)=\frac{\partial}{\partial\tilde{x}_{t_i}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)\Delta y_{t_j} \quad \text{(Eq. 3.23)}$$

$$\frac{d}{d(\frac{\partial v}{\partial x})}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)=\frac{\partial}{\partial\tilde{y}_{t_j}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)\ \Delta x_{t_i} \quad \text{(Eq. 3.24)}$$

$$\frac{d}{d(\frac{\partial v}{\partial y})}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)=\frac{\partial}{\partial\tilde{y}_{t_j}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)\Delta y_{t_j} \quad \text{(Eq. 3.25)}$$

From Eq. 3.20-Eq. 3.25, it is clear that the only quantities that need to be computed are $\frac{\partial}{\partial\tilde{x}_{t_i}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)$ and $\frac{\partial}{\partial\tilde{y}_{t_j}}G\left(\tilde{x}_{t_i},\tilde{y}_{t_j}\right)$. Since $\mathbf{P}_{\mathbf{tt}}^{(n)}=\mathbf{0}$, these derivatives correspond to the derivatives of the reference subset (at integer pixel locations) within the reference state.

Finally, $G\left(\mathbf{O}_{\boldsymbol{v}}+W\left(\boldsymbol{\Delta}\mathbf{X}_{\mathbf{ij}};\mathbf{P}_{\mathbf{tw}}^{(n)}\right)\right)$, like the derivatives, needs to be computed using the interpolated functions of the grayscale values of the corresponding image. In this case, the interpolation is needed given the fact that the points to be evaluated will be most likely in subpixel locations, that is, between pixels.

The program uses B-spline interpolation to approximate the image grayscale values through a linear combination of basis splines of degree n, $\varphi^n(x,y)$, which are scaled by means of their coefficients $c(k,l)$. These coefficients do not necessarily have to match the sample points.

Firstly, one-dimensional interpolation shall be introduced. The interpolated point can be computed as follows [17]:

$$g(x)=\sum_{k\in\mathbb{Z}}c(k)\varphi^n(x-k) \quad \text{(Eq. 3.26)}$$

A basis spline of degree n can be defined as follows:

$$\varphi^n(x) = \frac{1}{n!}\sum_{t=0}^{n+1}\binom{n+1}{t}(-1)^t\left(x - t + \frac{n+1}{2}\right)_+^n \qquad \text{(Eq. 3.27)}$$

where x_+^n is the one-sided power function [16] given by:

$$f(x) = \begin{cases} 0\ , & n = 0 \bigwedge x < 0 \\ 1/2\ , & n = 0 \bigwedge x = 0 \\ 1\ , & n = 0 \bigwedge x > 0 \\ x_+^0 x^n, & n > 0 \end{cases} \qquad \text{(Eq. 3.28)}$$

Figure 3.5 shows the representation of φ^n from degree 2 to 7, showing B-splines narrow support.

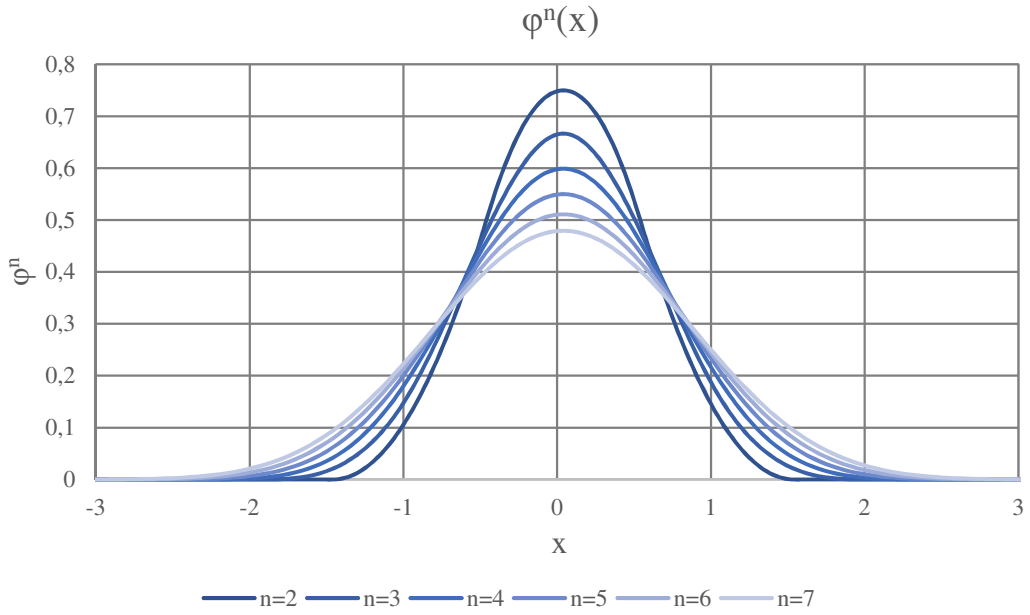


Figure 3.5 - B-splines from degree 2 to 7.

Any subpixel location $(x, y) = (\lfloor x \rfloor + \Delta x, \lfloor y \rfloor + \Delta y)$ can be computed by means of the next expression (this result is fully explained in 5.2), whenever $n = 5$ (otherwise, the lengths of the vectors of Δ need to be $n + 1$, so that the last term is Δ^n).

$$G(x,y) = [1 \quad \Delta y \quad \Delta y^2 \quad \Delta y^3 \quad \Delta y^4 \quad \Delta y^5][\mathrm{Z}] \begin{bmatrix} 1 \\ \Delta x \\ \Delta x^2 \\ \Delta x^3 \\ \Delta x^4 \\ \Delta x^5 \end{bmatrix} \tag{Eq. 3.29}$$

where [Z] is a matrix of values, different for each $(\lfloor x \rfloor, \lfloor y \rfloor)$ point, calculated for all the points before the beginning of the DIC analysis.

Using Eq. 3.29, the calculation of $G\left(\mathbf{O_v} + W\left(\mathbf{\Delta X_{ij}}; \mathbf{P_{tw}^{(n)}}\right)\right)$ is straightforward: the new coordinates of the current subset are computed with Eq. 3.2 and, for each point, the grayscale value is interpolated according to $(\Delta x, \Delta y)$.

The last quantities that need to be addressed are $\frac{\partial}{\partial \tilde{x}_{t_i}} G\left(\tilde{x}_{t_i}, \tilde{y}_{t_j}\right)$ and $\frac{\partial}{\partial \tilde{y}_{t_j}} G\left(\tilde{x}_{t_i}, \tilde{y}_{t_j}\right)$. Noting that $\frac{\partial}{\partial x}(G(x,y))$ from Eq. 3.29 is $\frac{\partial}{\partial \Delta x}(G(x,y))\frac{\partial \Delta x}{\partial x} = \frac{\partial}{\partial \Delta x}(G(x,y))$ (and equivalently for y), and that $\left(\tilde{x}_{t_i}, \tilde{y}_{t_j}\right)$ correspond to integer pixel locations, since $\mathbf{P_{tt}^{(n)}} = \mathbf{0}$, it follows that (again for $n = 5$):

$$\frac{\partial}{\partial \tilde{x}_{t_i}} G\left(\tilde{x}_{t_i}, \tilde{y}_{t_j}\right) = [1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0][\mathrm{Z}] \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{Eq. 3.30}$$

$$\frac{\partial}{\partial \tilde{y}_{t_j}} G\left(\tilde{x}_{t_i}, \tilde{y}_{t_j}\right) = [0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0][\mathrm{Z}] \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{Eq. 3.31}$$

4. Obtaining deformation data

4.1. Full field displacements

The procedure described throughout section 3 is valid for any material point within the region of interest (ROI). Nonetheless, computing each point separately is quite computationally expensive. In order to carry out these operations more efficiently, the reliability guided method (RG) is used [2,3,12]. Furthermore, it is worth noting that, provided that the strains are continuous enough, it is not necessary to compute every pixel within the ROI. Prior to the DIC analysis, the distance between subsets is chosen. This parameter, given a specific image resolution, has the greatest impact on the total computation time.

4.1.1. Reliability guided method

Before starting the RG scheme, the initial guess of the seed is computed, yielding $\mathbf{P}_{\mathbf{tw}}^{(0)}$, found by means of NCC. In fact, it is the only point which uses NCC and, therefore, the seed needs to be chosen carefully in case of large deformations or rotations, opting for those points which are deformed or rotated the least, as long as this is possible and the surface has an appropriate texture. The rest of the points use neighbouring information, since the strains are supposed continuous, leading to an efficient DIC analysis. The problem that arises from applying this procedure straightforwardly is that bad points (with cracks or inappropriate texture or light conditions) are going to spread the wrong initial guess to their neighbour points. Given that IC-GN method works around the local minimum, a solution may be found and assumed correct when it is not.

A strategy to overcome this problem, which gives RG method its name, is to compute the material points along the path with the highest ZNSSD coefficient. That is to say, after computing $\mathbf{P}_{\mathbf{tw}}^{*}$, the ZNSSD coefficient is calculated with this result for the surrounding points, as long as two conditions are met: to be a valid material point and to be a new point, that is, not yet computed. In order to be a valid material point, it needs to meet two additional conditions: to be inside the ROI, and to form a subset in a way that all the points within are contained in the ROI. It is clear that, regarding these last conditions, the second one includes the first one,

but it is worth stating both because it may be more efficient to exclude the point if the first one is not met and, therefore, not having to check all the points within the subset. Figure 4.1 shows this process step by step.

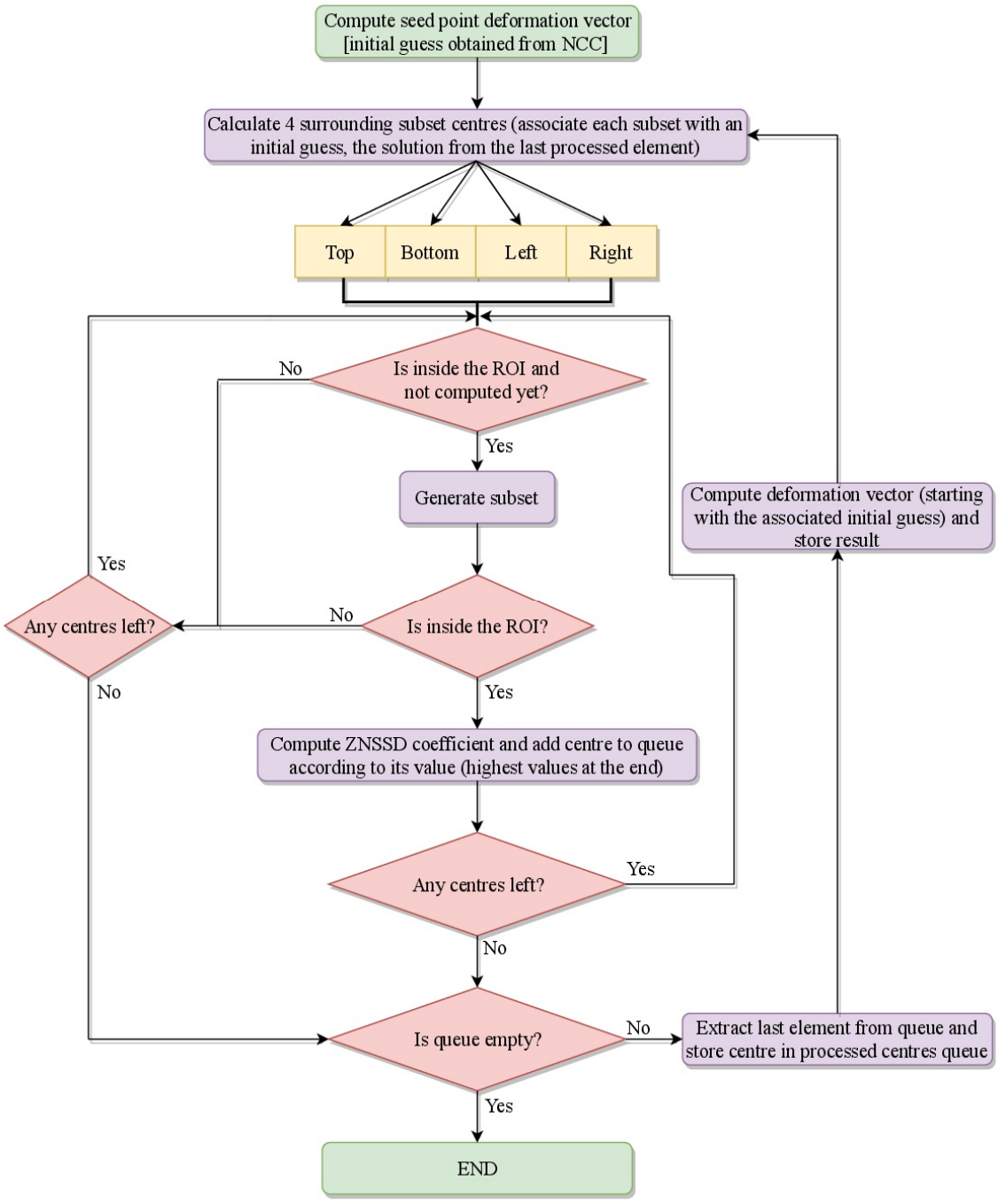


Figure 4.1 – Flow chart of the RG method.

Once all valid material points are computed, the next step is to calculate the strains. Although the deformation vectors $\mathbf{P}^*_{\mathbf{tw}}$ contain the displacement gradients, they are usually noisy, and must be smoothed. Therefore, the only relevant data from the DIC process are the u, v displacements. Before obtaining the full field strains, the missing points (due to the spacing between subsets or due to the fact that the subsets that exceed the ROI limits are excluded) are computed by means of cubic interpolation.

4.2. Full field strains

Given that differentiation is sensitive to noise, there is a need to smooth the displacement gradients. There are some techniques which use filtering in order to improve their smoothness, although in this project a different approach will be considered [2,3,13]. As previously mentioned, after computing the displacements and interpolating the missing material points (if the spacing of the subsets is greater than one pixel), two new arrays are formed, for u and v displacements.

Once again, an analogous kind of subset is defined, which may have a different size than the former subsets and, for convenience, it will be called a *window*. For each point in u and v matrices, a window is formed, being the point the centre of this window, within which the deformation is assumed constant. Therefore, the data points within the window can be fitted to a plane, the parameters of which are a constant (the adjusted displacement of the centre) and the gradients. If the coordinates of the centre are $(0,0)$, then both planes (for u and v displacements) are given by [2,3,4]:

$$u_{window}(x,y) = u_{centre}^{adjusted} + \frac{\partial u}{\partial x}x + \frac{\partial u}{\partial y}y \qquad \text{(Eq. 4.1)}$$

$$v_{window}(x,y) = v_{centre}^{adjusted} + \frac{\partial v}{\partial x}x + \frac{\partial v}{\partial y}y \qquad \text{(Eq. 4.2)}$$

Eq. 4.1 and Eq. 4.2 form the following over constrained system of equations.

$$\begin{bmatrix} 1 & x_0 - x_c & y_0 - y_c \\ \vdots & \vdots & \vdots \\ 1 & x_n - x_c & y_n - y_c \end{bmatrix} \begin{bmatrix} u_{centre}^{adjusted} \\ \dfrac{\partial u}{\partial x} \\ \dfrac{\partial u}{\partial y} \end{bmatrix} = \begin{bmatrix} u_0^* \\ \vdots \\ u_n^* \end{bmatrix} \quad \text{(Eq. 4.3)}$$

$$\begin{bmatrix} 1 & x_0 - x_c & y_0 - y_c \\ \vdots & \vdots & \vdots \\ 1 & x_n - x_c & y_n - y_c \end{bmatrix} \begin{bmatrix} v_{centre}^{adjusted} \\ \dfrac{\partial v}{\partial x} \\ \dfrac{\partial v}{\partial y} \end{bmatrix} = \begin{bmatrix} v_0^* \\ \vdots \\ v_n^* \end{bmatrix} \quad \text{(Eq. 4.4)}$$

In both Eq. 4.3 and Eq. 4.4, u_i^* and v_i^* are the previously obtained displacement data, and the subscripts $0 \dots n$ suggest two things: the first one, that not all points are valid to form a window, so n is equal or smaller than the size of the window squared minus 1; the second one, that the order of the points is not significant.

From the displacement gradient data, now any needed strain data may be calculated. In this project, Green strains are used. Being $\mathbf{F}$ the deformation gradient, the Green strain tensor is defined as follows:

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^T \cdot \mathbf{F} - \mathbf{I}) \quad \text{(Eq. 4.5)}$$

The term $\mathbf{F}^T \cdot \mathbf{F}$ completely eliminates the rigid body rotation. The explicit forms of Eq. 4.5 for the bidimensional case are:

$$E_{xx} = \frac{1}{2}\left[2\frac{\partial u}{\partial x} + \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2\right] \quad \text{(Eq. 4.6)}$$

$$E_{yy} = \frac{1}{2}\left[2\frac{\partial v}{\partial y} + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2\right] \quad \text{(Eq. 4.7)}$$

$$E_{xy} = \frac{1}{2}\left[\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} + \frac{\partial u}{\partial x}\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\frac{\partial v}{\partial y}\right] \quad \text{(Eq. 4.8)}$$

5. Calculation approach

5.1. Fast normalized cross-correlation

Given an undeformed subset S_t, the initial guess $\mathbf{P}_{\mathbf{tw}}^{(\mathbf{0})}$ is found by calculating C_{ZNCC} between S_t ($\mathbf{P}_{\mathbf{tt}} = \mathbf{0}$) and all possible $\tilde{S}_w$ which result from warping S_t with $\mathbf{P}_{\mathbf{tw}} = \{u \quad v \quad 0 \quad 0 \quad 0 \quad 0\}^T$, that is, a translation. Provided that there is not significant image scaling, rotation and perspective distortions, $\mathbf{P}_{\mathbf{tw}}^{(\mathbf{0})}$ will be $\mathbf{P}_{\mathbf{tw}}$ which yields the maximum C_{ZNCC}.

For convenience, Eq. 3.3 is redefined as follows:

$$C_{ZNCC}(u,v) = \frac{\sum_{(x,y)\in A}\left[G_t(x-u,y-v)-\bar{G}(S_t)\right]\left[G_w(x,y)-\bar{G}\left(\tilde{S}_w\right)\right]}{\sqrt{\sum_{(x,y)\in A}\left[G_t(x-u,y-v)-\bar{G}(S_t)\right]^2\sum_{(x,y)\in A}\left[G_w(x,y)-\bar{G}\left(\tilde{S}_w\right)\right]^2}} \quad \text{(Eq. 5.1)}$$

$$where\ A = \left[u+x_c-\left\lfloor\frac{L}{2}\right\rfloor, v+y_c-\left\lfloor\frac{L}{2}\right\rfloor\right] \times \left[u+L+x_c-\left\lfloor\frac{L}{2}\right\rfloor, v+L+y_c-\left\lfloor\frac{L}{2}\right\rfloor\right]$$

In Eq. 5.1, G_t and G_w mean that G is evaluated at the reference and current configuration, respectively, and L is the size of both S_t and $\tilde{S}_w$.

The following explanation aims to clarify this new definition of C_{ZNCC}. The point $(x_c - \lfloor\frac{L}{2}\rfloor, y_c - \lfloor\frac{L}{2}\rfloor)$ is the top left corner of S_t (see Figure 3.1), so A has the same dimensions as S_t, but moved according to (u, v). Consequently, $G_t(x-u, y-v)$ yields the same results regardless of $\mathbf{P}_{\mathbf{tw}}$. Actually, $\sum_{(x,y)\in A} G_t(x-u,y-v) = sum[G(S_t)]$, and S_t is often called a *template* in the literature, within the so-called *template matching* technique. On the other hand, $\sum_{(x,y)\in A} G_w(x,y)$ depends on (u, v). If $u = v = 0$, it means that $\sum_{(x,y)\in A} G_w(x,y)$ is evaluating the points at the current state which have the same coordinates as the points of S_t. For $u \neq 0$ or $v \neq 0$, the window defined by A moves across the current configuration. It is

clear that $(u, v) \in [-x_c + \lfloor \frac{L}{2} \rfloor, -y_c + \lfloor \frac{L}{2} \rfloor] \times [-x_c + \lfloor \frac{L}{2} \rfloor + H - L, -y_c + \lfloor \frac{L}{2} \rfloor + V - L]$, where (H, V) are the dimensions of the current state. Figure 5.1 aims to clarify the above.

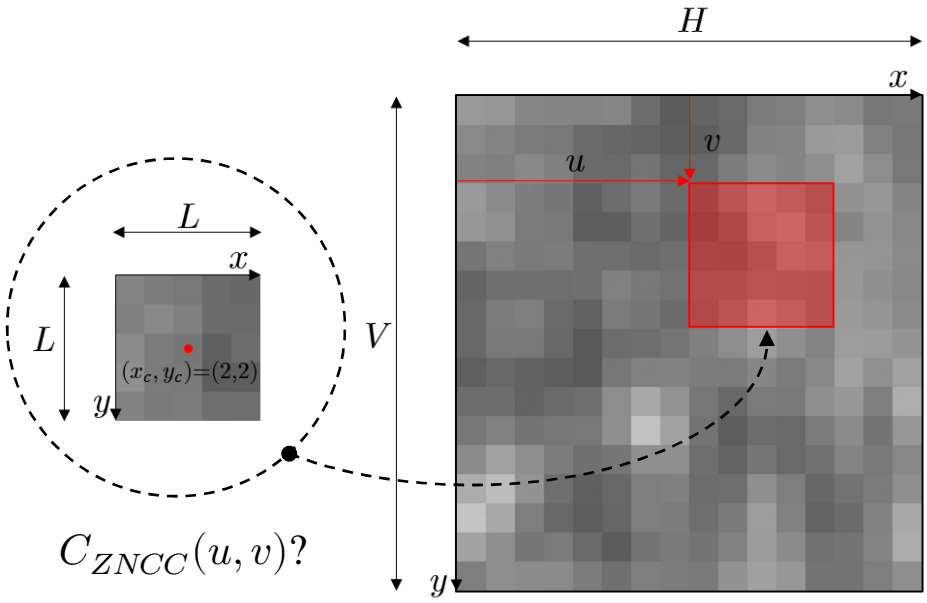


Figure 5.1 – Example of feature tracking. On the left, the reference subset (feature). On the right, the current configuration.

Eq. 5.1 is computationally expensive to calculate for each combination of (u, v); a faster algorithm, proposed by J. P. Lewis [6], will be used.

Defining $G'_t(x, y) = G_t(x, y) - \bar{G}(S_t)$ and $G'_w(x, y) = G_w(x, y) - \bar{G}\left(\tilde{S}_w\right)$, the numerator of Eq. 5.1 can be rewritten as $C^{num}_{ZNCC}(u, v) = \sum_{(x,y)\in A} G'_t(x - u, y - v) G'_w(x, y)$, which is a linear convolution of the current state $G'_w(x, y)$ with the reversed subset $G''_t(x, y) = G'_t(-x, -y)$.

$$C^{num}_{ZNCC}(u, v) = (G'_w * G''_t)(u, v) \qquad \text{(Eq. 5.2)}$$

Provided that both G'_w (of dimensions (H, V)) and G''_t (of dimensions (L, L)) are padded with zeros in such a way that their final dimensions are, at least, $(H + L - 1, V + L - 1)$, and applying the convolution theorem for the Discrete Fourier Transform (DFT), Eq. 5.2, gives:

$$\mathrm{DFT}[C^{num}_{ZNCC}] = \mathrm{DFT}[G'_w \circledast G''_t] = \mathrm{DFT}[G'_w]\mathrm{DFT}[G''_t] \qquad \text{(Eq. 5.3)}$$

Moreover, using the time-reversal and the complex conjugation properties of the DFT, which state, respectively, that $\mathrm{DFT}\{x[\langle -n \rangle_N]\} = X[\langle -k \rangle_N]$ and $\mathrm{DFT}\{x^*[n]\} = X^*[\langle -k \rangle_N]$, and given that $G \in [0,1] \in \mathbb{R} \rightarrow G = G^*$, it follows that $\mathrm{DFT}[G_t''(x,y)] = \mathrm{DFT}^*[G_t''(-x,-y)] = \mathrm{DFT}^*[G_t'(x,y)]$. Thus, Eq. 5.3 can be rewritten as follows, applying the inverse DFT to both sides:

$$C_{ZNCC}^{num} = \mathrm{DFT}^{-1}\left[\mathrm{DFT}[G_w'] \mathrm{DFT}^*[G_t'(x,y)]\right] \quad \text{(Eq. 5.4)}$$

Eq. 5.4 has the advantage of allowing a direct calculation of C_{ZNCC}^{num} using the original subset S_t, although it is also feasible to calculate it using Eq. 5.3 with an inverse copy of S_t. Additionally, rewriting the numerator of Eq. 5.1:

$$C_{ZNCC}^{num}(u,v) = \sum_{(x,y)\in A} G_w(x,y) G_t'(x-u, y-v) - \bar{G}\left(\tilde{S}_w\right) \sum_{(x,y)\in A} G_t'(x-u, y-v) \quad \text{(Eq. 5.5)}$$

Taking the second term of Eq. 5.5, it follows that:

$$\bar{G}\left(\tilde{S}_w\right) \sum_{(x,y)\in A} G_t'(x-u, y-v) =$$

$$\bar{G}\left(\tilde{S}_w\right) \left[\sum_{(x,y)\in A} G_t(x-u, y-v) - \sum_{(x,y)\in A} \bar{G}(S_t)\right] = \quad \text{(Eq. 5.6)}$$

$$\bar{G}\left(\tilde{S}_w\right) \left[\bar{G}(S_t) L^2 - L^2 \bar{G}(S_t)\right] = 0$$

In Eq. 5.6, $\sum_{(x,y)\in A} G_t(x-u, y-v) = sum[G(S_t)] = \bar{G}(S_t) L^2$, as stated above. Therefore, the numerator is reduced to:

$$C_{ZNCC}^{num}(u,v) = \sum_{(x,y)\in A} G_w(x,y) G_t'(x-u, y-v) \quad \text{(Eq. 5.7)}$$

Hence, the numerator can be calculated as follows:

$$C_{ZNCC}^{num} = \mathrm{DFT}^{-1}\left[\mathrm{DFT}[G_w]\mathrm{DFT}^{*}[G_t'(x,y)]\right] \quad \text{(Eq. 5.8)}$$

In order to calculate the denominator efficiently, it is important to notice that the windows that are calculated for different (u, v) overlap. As mentioned above, $\sum_{(x,y)\in A}\left[G_t(x-u, y-v) - \bar{G}(S_t)\right]^2$ can be precomputed. As for the expression $\sum_{(x,y)\in A}\left[G_w(x,y) - \bar{G}\left(\tilde{S}_w\right)\right]^2$, to avoid calculating the grayscale values of the same points, the integral (running sum) of its components can be computed prior to evaluating C_{ZNCC} at a certain displacement vector (u, v). Firstly, the expression needs to be decomposed as follows (for simplicity purposes, $G_w(x,y) = G_w$ and $\bar{G}\left(\tilde{S}_w\right) = \bar{G}_w$):

$$\sum_{(x,y)\in A}\left[G_w - \bar{G}_w\right]^2 = \sum_{(x,y)\in A} G_w^2 + \sum_{(x,y)\in A}\bar{G}_w^2 - 2\cdot\sum_{(x,y)\in A} G_w\bar{G}_w =$$

$$\sum_{(x,y)\in A} G_w^2 + \sum_{(x,y)\in A}\left[\frac{1}{L^2}\sum_{(x,y)\in A} G_w\right]^2 - 2\cdot\sum_{(x,y)\in A} G_w\left(\frac{1}{L^2}\sum_{(x,y)\in A} G_w\right) =$$

$$\sum_{(x,y)\in A} G_w^2 + \frac{1}{L^4}\sum_{(x,y)\in A}\left(\sum_{(x,y)\in A} G_w\right)^2 - \frac{2}{L^2}\sum_{(x,y)\in A} G_w\left(\sum_{(x,y)\in A} G_w\right) = \quad \text{(Eq. 5.9)}$$

$$\sum_{(x,y)\in A} G_w^2 + \frac{1}{L^4}\left(\sum_{(x,y)\in A} G_w\right)^2\sum_{(x,y)\in A} 1 - \frac{2}{L^2}\left(\sum_{(x,y)\in A} G_w\right)\sum_{(x,y)\in A} G_w =$$

$$\sum_{(x,y)\in A} G_w^2 - \frac{1}{L^2}\left(\sum_{(x,y)\in A} G_w\right)^2$$

Both terms in Eq. 5.9 can be calculated by means of the following running sums:

$$\begin{gathered} s(u,v) = f(u,v) + s(u-1,v) + s(u,v-1) - s(u-1,v-1) \\ s^2(u,v) = f^2(u,v) + s^2(u-1,v) + s^2(u,v-1) - s^2(u-1,v-1) \\ \text{with} \qquad s(u,v) = s^2(u,v) = 0 \qquad \text{when either} \qquad u,v < 0 \\ \text{for} \quad (u,v) \in \left[-x_c + \left\lfloor \frac{L}{2} \right\rfloor, -y_c + \left\lfloor \frac{L}{2} \right\rfloor\right] \times \left[-x_c + \left\lfloor \frac{L}{2} \right\rfloor + H - L, -y_c + \left\lfloor \frac{L}{2} \right\rfloor + V - L\right] \end{gathered} \tag{Eq. 5.10}$$

Finally, from the precomputed and stored values of the running sums s and s^2 in Eq. 5.10, Eq. 5.9 can be calculated as follows ($A = [A_1, A_2] \times [A_3, A_4]$, from Eq. 5.1):

$$\begin{gathered} \sum_{(x,y)\in A} G_w = s(A_3,A_4) + s(A_1-1,A_2-1) - s(A_1-1,A_4) - s(A_3,A_2-1) \\ \sum_{(x,y)\in A} G_w^2\,(x,y) = s^2(A_3,A_4) + s^2(A_1-1,A_2-1) - s^2(A_1-1,A_4) - s^2(A_3,A_2-1) \end{gathered} \tag{Eq. 5.11}$$

5.2. B-spline interpolation

5.2.1. Unidimensional

The main problem to be solved is to find the coefficients $c(k)$ which yield the integer samples, namely grayscale pixels (for example, pertaining to a row of the image), when evaluating the interpolation function $g(x)$ at these integer locations.

Eq. 3.26 is, in fact, the discrete convolution $g(x) = c(x) * \varphi^n(x)$ (at integer pixel locations x). Knowing that B-splines are symmetric, from Eq. 3.27 and Eq. 3.28 it follows that $\varphi^n(x)$ is 0 when $|x| > \lfloor \frac{n+1}{2} \rfloor$ ($\varphi^n(x)$ is a sequence of length $2\lfloor \frac{n+1}{2} \rfloor + 1$). This convolution can be converted into the circular convolution $c(x) \circledast \varphi^n(x)$ by means of zero-padding, provided that the final lengths of $c(x)$ and $\varphi^n(x)$ are, at least, the sum of their initial lengths minus 1. Thus, applying the DFT, $\mathrm{DFT}\{g(x)\} = \mathrm{DFT}\{c(x)\}\mathrm{DFT}\{\varphi^n(x)\}$, the coefficients can be calculated using Eq. 5.12 [8].

$$c(x) = \mathrm{DFT}^{-1}\left\{\frac{\mathrm{DFT}\{g(x)\}}{\mathrm{DFT}\{\varphi^n(x)\}}\right\} \qquad \text{(Eq. 5.12)}$$

In Eq. 5.12, the DFTs of $g(x)$ and $\varphi^n(x)$ are divided elementwise. The aforementioned condition on zero-padding will be true if $g(x)$ is padded with zeros up to $2\lfloor\frac{n+1}{2}\rfloor$, and $\varphi^n(x)$ up to the new length of $g(x)$ (the quantity $\lfloor\frac{n+1}{2}\rfloor$ is called half padding in this project).

The DFT is defined for non-negative sequence indices. Although $\varphi^n(x)$ could be treated as a sequence starting at $x = 0$, to preserve the original meaning of x, and hence to facilitate the recovery of the values, x is allowed to be negative. Moreover, since $\varphi^n(x)$ is symmetrical, $g(x)$ is defined analogously, being $x = 0$ the central point. Knowing that the set is treated as period of an infinite sequence under the DFT, the values of the negative positions are concatenated at the end of the interval. Therefore, being L the length of φ^n once padded (symmetrically) appropriately:

$$\{\varphi^n\} = \left\{\varphi(0), \dots, \varphi\left(\left\lceil\frac{L}{2}\right\rceil - 1\right), \varphi\left(-\left\lfloor\frac{L}{2}\right\rfloor\right), \dots, \varphi(-1)\right\} \qquad \text{(Eq. 5.13)}$$

Eq. 5.13 also applies, analogously, to $g(x)$. After computing the B-spline coefficients, the shifting should be reversed, splitting the sequence and concatenating the corresponding part at the end.

The program takes a step further regarding the interpolation process [2,3], for efficiency purposes, and to allow for a fast calculation of the derivatives. The interpolation function $g(x)$ can be seen as a dot product between the computed coefficients $c(k)$ and the B-spline values at $x - k$, although these last values must be computed for each x. Since a generalization would be complex, the following procedure is valid for degree 5, which is one of the B-spline orders chosen for the program.

The B-spline kernel of degree 5, found by solving Eq. 3.27, is:

$$\varphi^5(x) = \begin{cases} \frac{1}{120}x^5 + \frac{1}{8}x^4 + \frac{3}{4}x^3 + \frac{9}{4}x^2 + \frac{27}{8}x + \frac{81}{40}, & -3 \leq x \leq -2 \\ -\frac{1}{24}x^5 - \frac{3}{8}x^4 - \frac{5}{4}x^3 - \frac{7}{4}x^2 - \frac{5}{8}x + \frac{17}{40}, & -2 \leq x \leq -1 \\ \frac{1}{12}x^5 + \frac{1}{4}x^4 - \frac{1}{2}x^2 + \frac{11}{20}, & -1 \leq x \leq 0 \\ -\frac{1}{12}x^5 + \frac{1}{4}x^4 - \frac{1}{2}x^2 + \frac{11}{20}, & 0 \leq x \leq 1 \\ \frac{1}{24}x^5 - \frac{3}{8}x^4 + \frac{5}{4}x^3 - \frac{7}{4}x^2 + \frac{5}{8}x + \frac{17}{40}, & 1 \leq x \leq 2 \\ -\frac{1}{120}x^5 + \frac{1}{8}x^4 - \frac{3}{4}x^3 + \frac{9}{4}x^2 - \frac{27}{8}x + \frac{81}{40}, & 2 \leq x \leq 3 \\ 0, & elsewhere \end{cases} \quad \text{(Eq. 5.14)}$$

For convenience, the piecewise function $\varphi^5(x)$ will be redefined in such a way that the new intervals in which the polynomials are defined become $[-3-2[$, $[-2,-1[$, $[-1,0[$, $[0,1[$, $[1,2[$ and $[2,3[$. Noting that $x = \lfloor x \rfloor + \Delta x$, where Δx is the distance between x and its floor, the modified B-spline kernel of degree 5 becomes:

$$\varphi^5(\lfloor x \rfloor + \Delta x) =$$

$$\begin{cases} \frac{1}{120}\Delta x^5, & -3 \leq x < -2 \\ -\frac{1}{24}\Delta x^5 + \frac{1}{24}\Delta x^4 + \frac{1}{12}\Delta x^3 + \frac{1}{12}\Delta x^2 + \frac{1}{24}\Delta x + \frac{1}{120}, & -2 \leq x < -1 \\ \frac{1}{12}\Delta x^5 - \frac{1}{6}\Delta x^4 - \frac{1}{6}\Delta x^3 + \frac{1}{6}\Delta x^2 + \frac{5}{12}\Delta x + \frac{13}{60}, & -1 \leq x < 0 \\ -\frac{1}{12}\Delta x^5 + \frac{1}{4}\Delta x^4 - \frac{1}{2}\Delta x^2 + \frac{11}{20}, & 0 \leq x < 1 \\ \frac{1}{24}\Delta x^5 - \frac{1}{6}\Delta x^4 + \frac{1}{6}\Delta x^3 + \frac{1}{6}\Delta x^2 - \frac{5}{12}\Delta x + \frac{13}{60}, & 1 \leq x < 2 \\ -\frac{1}{120}\Delta x^5 + \frac{1}{24}\Delta x^4 - \frac{1}{12}\Delta x^3 + \frac{1}{12}\Delta x^2 - \frac{1}{24}\Delta x + \frac{11}{20}, & 2 \leq x < 3 \\ 0, & elsewhere \end{cases} \quad \text{(Eq. 5.15)}$$

Knowing that B-splines support for odd degrees is $[-\frac{n+1}{2}, \frac{n+1}{2}]$, a point $x = \lfloor x \rfloor + \Delta x$ can be interpolated using Eq. 3.26 limiting the range of k to $[\lceil x \rceil - \frac{n+1}{2}, \lfloor x \rfloor + \frac{n+1}{2}]$, which can be assumed $[\lfloor x \rfloor + 1 - \frac{n+1}{2}, \lfloor x \rfloor + \frac{n+1}{2}]$. Therefore, Eq. 3.26 takes the following form, for $n = 5$.

$$g(x) = \sum_{k=\lfloor x \rfloor - 2}^{\lfloor x \rfloor + 3} c(k)\varphi^5(\lfloor x \rfloor + \Delta x - k) = \begin{bmatrix} c(\lfloor x \rfloor - 2) \\ c(\lfloor x \rfloor - 1) \\ c(\lfloor x \rfloor) \\ c(\lfloor x \rfloor + 1) \\ c(\lfloor x \rfloor + 2) \\ c(\lfloor x \rfloor + 3) \end{bmatrix} \begin{bmatrix} \varphi^5(\Delta x + 2) \\ \varphi^5(\Delta x + 1) \\ \varphi^5(\Delta x) \\ \varphi^5(\Delta x - 1) \\ \varphi^5(\Delta x - 2) \\ \varphi^5(\Delta x - 3) \end{bmatrix} = \quad \text{(Eq. 5.16)}$$

$$\begin{bmatrix} c(\lfloor x \rfloor - 2) \\ c(\lfloor x \rfloor - 1) \\ c(\lfloor x \rfloor) \\ c(\lfloor x \rfloor + 1) \\ c(\lfloor x \rfloor + 2) \\ c(\lfloor x \rfloor + 3) \end{bmatrix} \left\{ [\gamma] \begin{bmatrix} 1 \\ \Delta x \\ \Delta x^2 \\ \Delta x^3 \\ \Delta x^4 \\ \Delta x^5 \end{bmatrix} \right\}^{flipped}$$

where $[\gamma] = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \frac{1}{120} \\ \frac{1}{120} & \frac{1}{24} & \frac{1}{12} & \frac{1}{12} & \frac{1}{24} & -\frac{1}{24} \\ \frac{13}{60} & \frac{5}{12} & \frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} & \frac{1}{12} \\ \frac{11}{20} & 0 & -\frac{1}{2} & 0 & \frac{1}{4} & -\frac{1}{12} \\ \frac{13}{60} & -\frac{5}{12} & \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} & \frac{1}{24} \\ \frac{11}{20} & -\frac{1}{24} & \frac{1}{12} & -\frac{1}{12} & \frac{1}{24} & -\frac{1}{120} \end{bmatrix}$, and $\{...\}^{flipped}$ denotes the result of flipping the results within, that is, that the first element becomes the last, and so on.

Concluding, to interpolate any point $x = \lfloor x \rfloor + \Delta x$, $c(k)$ is shifted $\lfloor x \rfloor$ positions left, and a dot product is performed with the modified B-spline kernel, which is calculated by means of the product between the kernel coefficients and the vector of Δx, flipping the result.

5.2.2. Bidimensional

Similarly to the 1-D B-spline interpolation previously explained, the 2-D interpolation function $g(x, y)$ is defined as follows:

$$g(x, y) = \sum_{k \in \mathbb{Z}} \sum_{l \in \mathbb{Z}} c(k, l)\varphi^n(x - k)\varphi^n(x - l) \quad \text{(Eq. 5.17)}$$

Due to the B-spline separability property, the coefficients $c(k,l)$ can be computed using two one-dimensional steps: the first one takes each row of the ROI and computes intermediate coefficients, creating a new padded bi-dimensional array; the second one takes each column of this new array and, analogously, computes the final matrix of coefficients $[KL]$. Carrying out the computation in this way is faster than, for example, using 2-D Fourier transforms.

Finally, following the same steps as in the previous section, the 2-D interpolation function becomes:

$$g(x,y) = \{[1 \quad \Delta y \quad \Delta y^2 \quad \Delta y^3 \quad \Delta y^4 \quad \Delta y^5][\gamma]^T\}^{fl.}[KL]^{shft.}\left\{[\gamma]\begin{bmatrix}1\\ \Delta x\\ \Delta x^2\\ \Delta x^3\\ \Delta x^4\\ \Delta x^5\end{bmatrix}\right\}^{fl.} \qquad \text{(Eq. 5.18)}$$

where $[KL]^{shft.} = \begin{bmatrix} c(-2+\lfloor x\rfloor, -2+\lfloor y\rfloor) & \cdots & c(3+\lfloor x\rfloor, -2+\lfloor y\rfloor) \\ \vdots & \ddots & \vdots \\ c(-2+\lfloor x\rfloor, 3+\lfloor y\rfloor) & \cdots & c(3+\lfloor x\rfloor, 3+\lfloor y\rfloor) \end{bmatrix}$.

The 2-D interpolation function can also be written as follows:

$$g(x,y) = [1 \quad \Delta y \quad \Delta y^2 \quad \Delta y^3 \quad \Delta y^4 \quad \Delta y^5]\{[\gamma]^T\}^{HF}[KL]^{shft.}[\gamma]^{VF}\begin{bmatrix}1\\ \Delta x\\ \Delta x^2\\ \Delta x^3\\ \Delta x^4\\ \Delta x^5\end{bmatrix} \qquad \text{(Eq. 5.19)}$$

where HF and VF stand for horizontally flipped and vertically flipped, respectively.

Computing $[\mathrm{Z}] = \{[\gamma]^T\}^{HF}[KL]^{shft.}[\gamma]^{VF}$ in the first place, for every pixel of both the reference image and the current image, will save some computational costs.

6. Computational approach

The programming language used to implement the algorithm and facilitate the interaction with the user has been Python. Python allows writing code quite similar to human language in an agile way, unlike others of lower level. On the contrary, it is much less efficient, in its native form, than other programming languages such as C.

One of the main advantages, which follows from the above, is that it allows testing the different parts of the DIC algorithm without spending excessive time in programming. Once the partial steps have been verified, however, the main disadvantage is that part of the simplicity is lost, given that some techniques and external libraries must be used beyond the standard Python tools to reach reasonable levels of speed.

In this section, the basis on which the code has been developed is presented, with the aim of indicating the necessary libraries and also to show the so-called vectorization technique that, although it may seem unnatural, is much more efficient that plain Python code.

6.1. External Python libraries

6.1.1. NumPy

NumPy [11] is the most important library that has been used in this project. It is a scientific package that offers a set of tools for array calculations. It is written almost completely in C, which allows to combine the speed of this language with the simplicity of Python. In addition, it offers a set of broadcasting functions, which means that their corresponding operations can be performed using C loops, as explained in 6.2.

The object with which most functions work is the *ndarray*, a collection of items of the same type, which are accessed using indexing or slicing. The indexing can be more or less sophisticated according to the sought efficiency and simplicity. By default, *float64* datatype is used throughout the program.

These arrays have a particular dimension, usually fixed, associated with them. For example, the images that have to be loaded prior to the DIC analysis are saved as an *ndarray* of two dimensions. Each of these dimensions has the size of the corresponding side of the image.

6.1.2. SciPy

The SciPy [15] library offers additional functionalities to those of NumPy, along with which form the essential scientific pack for Python. Among them, there are tools for statistical analysis, signal processing, optimization and other numerical algorithms. In this project, the submodules that have been used are *scipy.special*, *scipy.fftpack* and *scipy.interpolate.griddata*, for factorial calculation, Fast Fourier Transform and cubic interpolation, respectively. Like NumPy, it is written mainly in C.

6.1.3. OpenCV

OpenCV (Open Source Computer Vision Library) [10] is a library that consists of a set of tools and algorithms for computer vision and machine learning. None of these features is used in this project, the library is only necessary to load the images. An advantage to be highlighted is that there is no need to perform any additional operations, thanks to the fact that the images are saved directly as *ndarray* objects of two dimensions. A point worth noting is that these arrays must be divided by 255, since the intensity of the pixels falls within the range $[0,255]$, but later operations using *float64* convert these integer images into float images, which are valid within the $[0,1]$ range.

6.1.4. Matplotlib

Matplotlib [5] is used in this project mainly for post-processing plots, but also for facilitating the interaction with the user in the definition of the DIC parameters, by showing the loaded images. It is a plotting library that generates a wide variety of interactive figures, and adapts perfectly to NumPy, which means that *ndarray* objects can be plotted without having to perform too many operations.

6.1.5. Cython

Cython [1] is a special library designed to improve the performance of Python by converting the code into C language. It also supports calling C functions from Python modules and declaring static variables. Although the efficiency of the code can be improved by simply converting it into C, the advantages of using this library are really noticed once certain changes are made in the code, such as specifying the variable type.

Most of the tools of this library have not yet been explored, but it is necessary to convert the code into C to generate an executable file, this is why Cython is part of the list.

6.1.6. PyInstaller

In this project, the combination of Cython with PyInstaller aims to generate an executable, stand-alone file that does not depend on any of the aforementioned libraries, so that the program can be run from any Windows device. For this particular code, the improvement in performance has not been significant with respect to running the original code from the console.

6.2. Vectorization

Python loops are remarkably slower than C loops, and for this reason they should be avoided as much as possible in applications that require many calculations made within an iterative structure. With regard to the DIC algorithm, there are many operations that cannot be performed by means of basic matrix calculations, and the most natural method to carry them out is using loops. The following example shows a comparison of the speed when using a default NumPy function to add all the elements of a random $[10000,10000]$ matrix and when doing the same operation by means of Python iterations.

```
>>> import time
>>> import numpy as np
>>> MATRIX=np.random.random((10000,10000))
>>> def fast(M):
...     now=time.time()
...     sum=M.sum()
...     later=time.time()
...     return later-now
...
>>> def slow(M):
...     now=time.time()
...     sum=0
...     for j in range(10000):
...             for i in range(10000):
...                     sum+=M[j,i]
...     later=time.time()
...     return later-now
...
>>> fast(MATRIX)
0.10938000679016113
>>> slow(MATRIX)
18.337414264678955
>>> slow(MATRIX)/fast(MATRIX)
180.65395463601132
```

Box 6.1 – Comparison of the speed between vectorized operations and Python loops.

Box 6.1 shows that, in this case, the vectorized operation is 180 times faster than the iteration. In fact, this type of operations are merely loops executed at C level, which is why the speed increases very significantly. Unfortunately, most of the operations needed for this project do not have a clear equivalent in NumPy, but finally they have been transformed into a set of much more efficient vectorized procedures, improving the overall speed of the program. The following example shows one of the many sections of the code that uses this technique, specifically the functions `precomp`, `coef_2D` and `ck_slice`.

```
import numpy as np
import cv2
from Interpolation_1D import *
import time

def ck_slice(CK,x,y,rows_ck,cols_ck,n,ADD_X,ADD_Y):

       return CK[int(np.floor(rows_ck/2))-np.int((n+1)/2)+y+1+ADD_Y,
       np.int(np.floor(cols_ck/2))-np.int((n+1)/2)+x+1+ADD_X]

def coef_2D(image,n,num):

       rows=image.shape[0]
       cols=image.shape[1]
       CK=np.zeros((rows+2*num,cols+2*num))

       CK[num:-num,:]=np.apply_along_axis(coef,1,image,n,num)
       CK[:,:]=np.apply_along_axis(coef,0,CK[num:-num,:],n,num)

       return CK

def precomp(IMAGE,n,num):

       ck=coef_2D(optim_fft(IMAGE,num),n,num)

       R=IMAGE.shape[0]
       C=IMAGE.shape[1]
       rows_ck=ck.shape[0]
       cols_ck=ck.shape[1]

       x_rep=np.arange(C)
       y_rep=np.arange(R).reshape(R,1)

       X=np.tile(np.repeat(x_rep,(n+1)**2),(R,1))
       Y=np.tile(y_rep,(1,C*(n+1)**2))

       ADD=np.arange(n+1)
       ADD_X=np.tile(ADD,C*R*(n+1)).reshape((R,C*(n+1)**2))
       ADD_Y=np.tile(np.repeat(ADD,n+1),C*R).reshape((R,C*(n+1)**2))

       SLICED_CK=ck_slice(ck,X-np.int(np.floor(C/2)),Y-
np.int(np.floor(R/2)),rows_ck,cols_ck,n,ADD_X,ADD_Y).reshape((R,C,n+1,n+1))

       del ADD_X,ADD_Y,X,Y

       INNER_MATRIX=np.einsum('kl,ijlp->ijkp',KERNEL[n][1],SLICED_CK)

       del SLICED_CK
       INNER_MATRIX=np.einsum('ijkp,pl->ijkl',INNER_MATRIX,KERNEL[n][2])

       return INNER_MATRIX
```

Box 6.2 – Functions `ck_slice`, `coef_2D` and `precomp` showing the use of vectorization.

The function `precomp` computes $[\mathrm{Z}] = \{[\gamma]^T\}^{HF}[KL]^{shft.}[\gamma]^{VF}$ from Eq. 5.19 prior to the DIC analysis, for each pixel of `IMAGE`. Therefore, the resulting array is of dimensions $(V, H, n+1, n+1)$, where V and H are the vertical and horizontal lengths, respectively, and n is the degree of interpolation. The line by line explanation is the following.

- `ck` is the (V, H) array of B-spline interpolation coefficients, which is computed by means of the `coef_2D` function above. This last function calls the function `coef` from another module of the code, which calculates the unidimensional coefficients. `np.apply_along_axis` is already a vectorized function, which takes as parameters a unidimensional function (in this case, `coef`), the axis to which it will be applied and the parameters of the unidimensional function.
- `R`, `C`, `rows_ck` and `cols_ck` are, respectively, the rows and columns of the image and the rows and columns of the coefficients array. They are different given that, to obtain the coefficients, the image must be padded.
- `x_rep` and `y_rep` are, respectively, a column of length V and a row of length H, the values of which start at 0 and end at $V-1$ and $H-1$.
- `X`, `Y`, `ADD_X` and `ADD_Y` serve as unidimensional indices. The example below shows the use of these matrices.
- `SLICED_CK` is an array returned by `ck_slice`. Each $(n+1, n+1)$ subarray of `SLICED_CK` corresponds to $[KL]^{shft.}$.
- `del` deletes the unnecessary arrays, because they usually take up a lot of memory.
- `INNER_MATRIX` is a $(V, H, n+1, n+1)$ array, each subarray of which corresponds to $\{[\gamma]^T\}^{HF}[KL]^{shft.}[\gamma]^{VF}$ for a particular integer pixel location (x, y) within $[0, H-1] \times [0, V-1]$.

To form `SLICED_CK` by means of vectorization, the underlying idea is to make $(n+1, n+1)$ windows over `ck` for each pixel, all at once, using fancy indexing techniques. For example, taking $n = 1$ and a tiny (2,3)-pixel image[1], `X`, `Y`, `ADD_X` and `ADD_Y` take the following forms:

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 \end{bmatrix} \qquad \text{(Eq. 6.1)}$$

[1] None of the values in this example are valid, but the idea remains the same. Small values have been chosen for space and clarity purposes.

$$Y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad \text{(Eq. 6.2)}$$

$$ADD\_X = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad \text{(Eq. 6.3)}$$

$$ADD\_Y = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \quad \text{(Eq. 6.4)}$$

Since the four arrays have the same shape, if the function `ck_slice` is called passing them as parameters, it will operate their elements in groups of four, for each index. This is the so-called NumPy broadcasting. The purpose of `X` and `Y` is to serve as indices. Given that $[KL]^{shft.}$ has the shape $(2,2)$, if the values of `ck` are accessed individually, 4 indices are needed. Whereas `X` and `Y` provide the integer pixel locations of the original image, `ADD_X` and `ADD_Y` provide, for each $[KL]^{shft.}$ that needs to be calculated, the values that have to be added to the upper left margin to extract the $(2,2)$ array from `ck`. The following figure aims to clarify the above, for a single material point of the image. Actually, the reshaping operation is not performed until all the windows of the coefficients matrix have been extracted, but the idea remains the same (in this case, the reshaped matrix is one of the $(2,2)$ subarrays of the $(2,3,2,2)$ precomputed interpolation array).

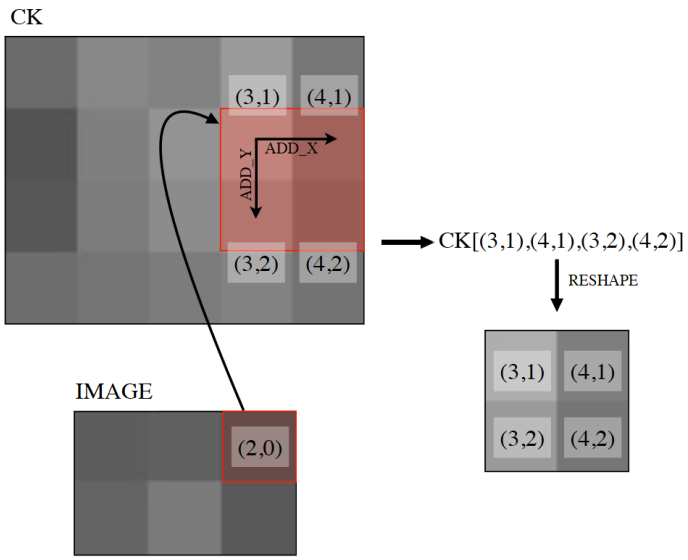


Figure 6.1 – Example of coefficients extraction for a single material point using vectorization, for $n = 1$ and a $(2,3)$-pixel image.

Finally, in order to obtain `INNER_MATRIX`, the B-spline kernels must be multiplied. Obviously, this operation cannot be performed directly, since these kernels have the shape $(n+1, n+1)$, while `SLICED_CK` have the shape $(V, H, n+1, n+1)$. A possible solution, which belongs to the group of vectorized functions of NumPy as well, is to use Einstein summation along the right axes. This type of operation has been used in almost all of the other functionalities of the code. A graphic example of this procedure, following the previous example, is presented below.

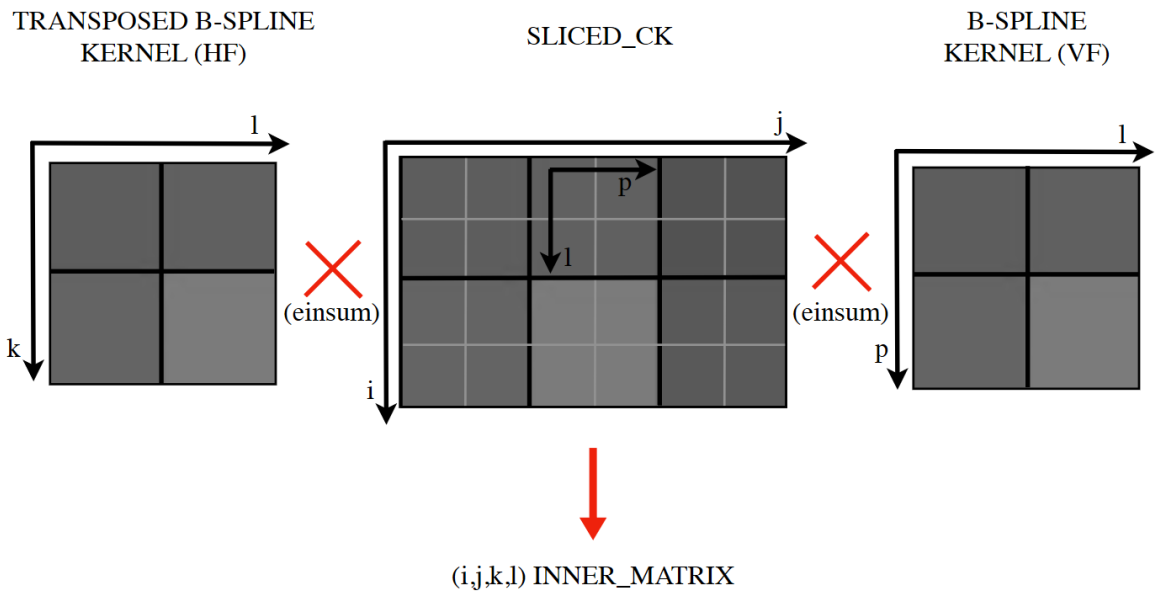


Figure 6.2 – Example of Einstein summation for $n = 1$ and a (2,3)-pixel image.

7. The program

7.1. Functions and modules

The code of the program is distributed in different modules, which approximately correspond to the different blocks that have been explained so far. The following diagram shows the relationship between them (the arrows go from father to son).

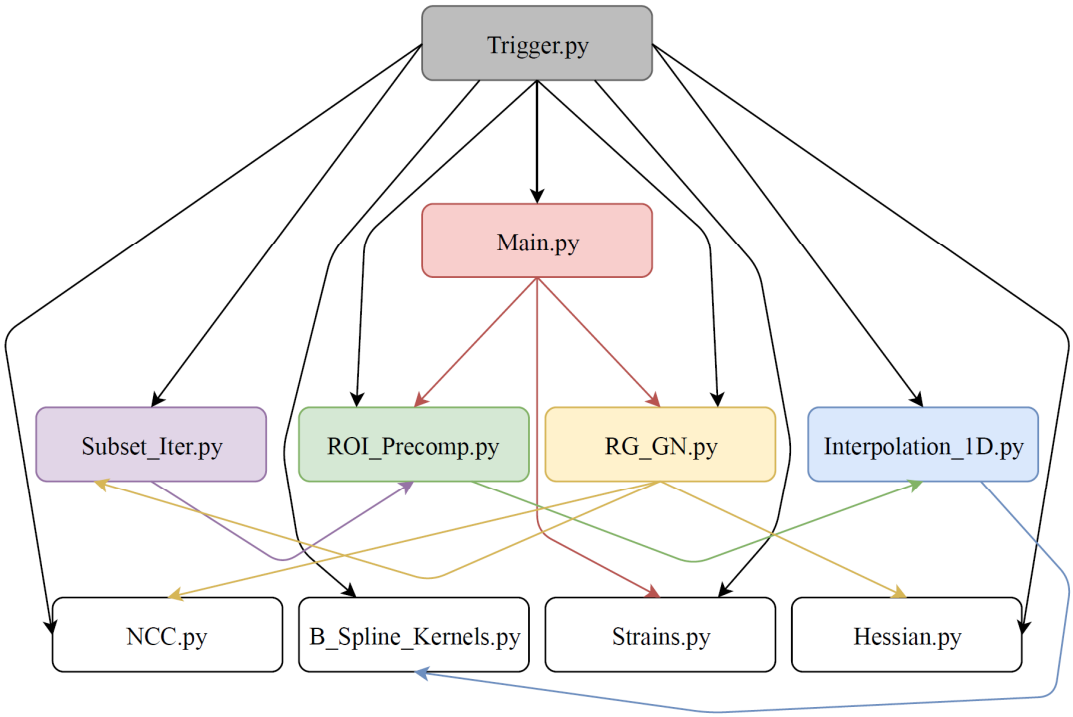


Figure 7.1 – Diagram of relationships between modules.

Trigger.py contains all the imports of all the necessary libraries so that the code can be converted into C, in case generating an executable file is desired. *Main.py* connects the different modules and allows the interaction via console. Moreover, it performs some intermediate operations so that functions are called passing the appropriate parameters. For example, after computing the displacements of the material points, the missing pixels due to the subset spacing are interpolated, generating a continuous image within the ROI. The functions of each of the rest of the modules are explained in the following tables.

Subset_Iter.py
`current_pos(X,Y,p0,p1,p2,p3,p4,p5,ctr_x,ctr_y)`
It calculates the current subset positions, that is, the new non-integer positions. It does so from the deformation vector $\mathbf{P} = (p0, p1, p2, p3, p4, p5)$ applied to a point (X, Y) belonging to a subset with centre (ctr_x, ctr_y). This function has been completely vectorized (to allow broadcasting, the components of $\mathbf{P}$ are passed split).
`DELTA_P(R_SUB,ctr,p_old,in_mat,DER_X,DER_Y,Hessian,n)`
Corresponds to a single iteration. It finds the vector $\mathbf{P_{tt}}$ of the reference subset R_SUB with centre ctr, from the vector $\mathbf{P_{tw}}$ (p_old) found in the previous iteration. To do so, it needs the $(V, H, n+1, n+1)$ precomputed matrix, as well as its derivatives DER_X and DER_Y, along with the $Hessian$. n is the degree of interpolation, a parameter shared by many other functions. It calls `fwd_bwd_6` to find $\mathbf{P_{tt}}$.
`fwd_bwd_6(A,b,is_fwd)`
Performs a forward or backward substitution to solve $\mathbf{Ax} = \mathbf{b}$, according to the Boolean value is_fwd.

Table 7.1 – Functions in Subset_Iter.py.

ROI_Precomp.py
`ck_slice(CK,x,y,rows_ck,cols_ck,n,ADD_X,ADD_Y)`
Returns a value of the subarray $CK[int(np.\,floor(rows\_ck/2)) - np.\,int((n+1)/2) + y + 1 + ADD\_Y, np.\,int(np.\,floor(cols\_ck/2)) - np.\,int((n+1)/2) + x + 1 + ADD\_X]$. (x, y) is referenced to the center of the image from which it comes, and is later associated with the corresponding coordinate of the coefficients matrix `ck`, since they have different dimensions. The quantity $(ADD\_X + 1 - np.\,int((n+1)/2), ADD\_Y + 1 - np.\,int((n+1)/2))$ is added to obtain, point by point, the full $(n+1, n+1)$ matrix.
`coef_2D(image,n,num)`
It finds the B-spline interpolation coefficients of the bidimensional array $image$. num is the value of the half padding, and is a parameter shared by many other functions.

`optim_fft(img,num)`
It resizes the image img into another image whose dimensions are the next power of two of the original ones. By doing so, the DFT calculations are much faster, given the nature of the Cooley-Tukey FFT algorithm.
`precomp(IMAGE,n,num)`
It computes, in a vectorized way, the $(V, H, n+1, n+1)$ array of Z subarrays.
`precomp_RAM(IMAGE,n,num,steps,overlap)`
This function has not yet been fully tested. It computes the same as the previous function, but splitting the image into smaller images, in order to reduce the RAM peak.
`find_cut_in_mat(in_mat,x,y,n)`
It returns, in a vectorized way, the necessary points of the $(V, H, n+1, n+1)$ precomputed array in_mat for each point within a subset whose positions (x, y) are non-integer (although it is used for reference subsets as well).
`fast_intpol_2D_precomp(x,y,cut_in_mat,n)`
Performs the interpolation of a subset whose points are given by the ordered vectors $\boldsymbol{x}, \boldsymbol{y}$, using the Z matrix cut_in_mat.
`derivative(ROI,n,num,is_x,PRE)`
Since the derivatives in the IC-GN algorithm are evaluated at integer pixel locations, they can be computed before starting the iterations. This function computes the derivative with respect to x or y according to the Boolean value is_x. ROI is passed just to know the dimensions of the array. The derivatives come from the $(V, H, n+1, n+1)$ precomputed array PRE.

Table 7.2 – Functions in ROI_Precomp.py.

RG_GN.py
`ZNCC(a,b)`
It returns the C_{ZNCC} coefficient between subsets a and b.
`RG(REF,SEED,SEED_CTR,CUR,ERROR,DEGREE_INTPOL,PADDING,CUTOFF,STEPS, OVERLAPPING,NCC_WINDOW_SHAPE,NCC_WINDOW_MOVE,SUB_SPACE,FILTER, SMOOTH_WINDOW)`
This function is responsible for calling all the others involved in the DIC algorithm (from the initial solution of the seed to the final calculation of the last subset), and manages the

iterative scheme of all the subsets using the RG method. It also calculates, before the beginning of the iterations, a list with all of the valid subsets. When the calculation of a subset ends, it decides whether the neighbouring subsets are added to the stack or not. REF is the reference image, $SEED$ is the subset of the seed, $SEED\_CTR$ are the seed coordinates, CUR is the current image, $ERROR$ is the maximum allowed error, $DEGREE\_INTPOL$ is the degree of interpolation, $PADDING$ is the value of the half padding, cutoff is the maximum number of iterations allowed per subset, both $STEPS$ and $OVERLAPPING$ have to do with `precomp_RAM` (still under development), NCC_WINDOW_SHAPE are the dimensions of the window in which the initial guess will be searched (it is used to reduce the computational cost, and also to facilitate the search), NCC_WINDOW_MOVE are the upper left coordinates of the search window, SUB_SPACE is the subset spacing, $FILTER$ is the image which indicates where the ROI is, and $SMOOTH\_WINDOW$ is the strain window size.
`excluded(CTR,size,img,expanded_mod_filt)`
Decides whether a subset with centre CTR and dimensions $(size, size)$ is a valid subset to compute, according to the original image img and the expanded filter $expanded\_mod\_filt$.
`gen_excluded_alfa(img,filter_img)`
It shows an overlap of the original image img with the filter $filter\_img$ applied, that is, the ROI.

Table 7.3 – Functions in RG_GN.py.

Interpolation_1D.py
`pad(org,method,num)`
It pads org with num ones or zeros (according to $method$) on each side.
`move(array)`
It splits $array$ in half, and adds the first part at the end. It is used to recover the data after using the DFT.
`delta_gen(n,delta)`
It returns the first $n+1$ elements of the vector $[1, delta, delta^2, delta^3, delta^4, delta^5]$.

Table 7.4 – Functions in Interpolation_1D.py.

NCC.py
`ncc(image,feature,win_shape,win_posXY)`
It performs the fast normalized cross-correlation algorithm. win_shape and win_posXY limit the subset $feature$ search within $image$. It returns a matrix of approximate dimensions win_shape, each point of which is the value of C_{ZNCC}.
`max_ncc(ncc_matrix,win_posXY)`
It returns the coordinates of the point with the highest C_{ZNCC} among all ncc_matrix points.
`int_displacements(point,ctr,l)`
It returns the integer displacements from the reference configuration (seed centre ctr) to the current configuration ($point + l$ centre).

Table 7.5 – Functions in NCC.py.

B_Spline_Kernels.py
This file contains the interpolation kernels, for both degrees 3 and 5, expressed in terms of (x, y) and $(\Delta x, \Delta y)$, and the transposed kernel in terms of $(\Delta x, \Delta y)$.

Table 7.6 – Contents of B_Spline_Kernels.py.

Strains.py
`pre_strain(P_arr,space,dim)`
The purpose of this function is to extract the obtained displacement values from P_arr to form the U and V displacement matrices, according to the subset spacing $space$ and the dimensions dim of the original image.
`strains(P_arr,space,dim,filt,window)`
It fits the displacement matrices U and V to square planes of dimensions $(window, window)$ to smooth the data wherever this is possible (according to the ROI image $filt$). Then, it interpolates the missing points due to the fact that a subset spacing has been defined.

Table 7.7 – Functions in Strains.py.

Hessian.py
`Hessian(SUB,DER_X,DER_Y,ctr)`
This function has been fully vectorized. It computes the Hessian of a reference subset SUB with centre ctr, using the derivatives of the interpolation precomputed arrays DER_X and DER_Y. The result is a $(6,6)$ matrix.

Table 7.8 – Functions in Hessian.py.

7.2. User interface

This section it not only intended to show the graphic aspect of the program, but also the entire procedure that the user must follow to execute it correctly. As a result, all the concepts explained so far can be better understood. It will be assumed that the executable file is run, despite the fact that, as long as Python and all the dependencies are installed, the program can be run executing Main.py as well. It only works on 64-bit Windows operating systems (the 64-bit libraries allow to manage arrays larger than 4 GB).

Before entering the parameters, a set of recommendations about them are shown.

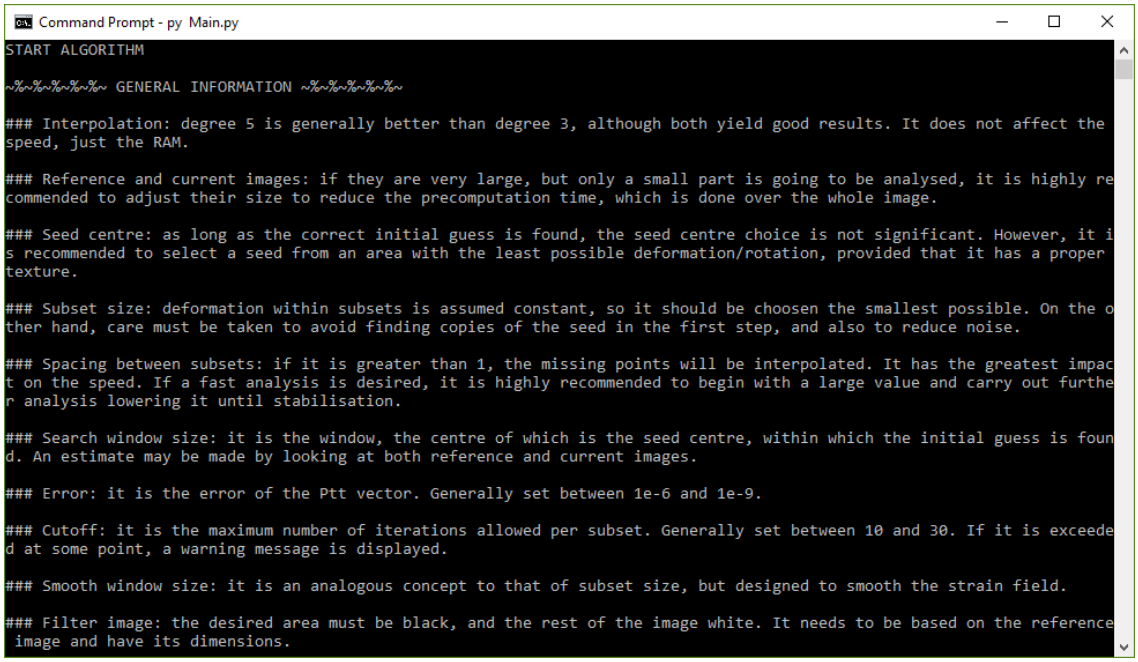


Figure 7.2 – First lines of the program.

Then, the degree of interpolation and the names of the reference image and the current image, respectively, must be entered. These images must be located in the same folder as the executable file. Otherwise, the full path must be given. For this example, two computer generated images will be used, which can be found in [2]. After entering the name or the path of the images, they are displayed separately.

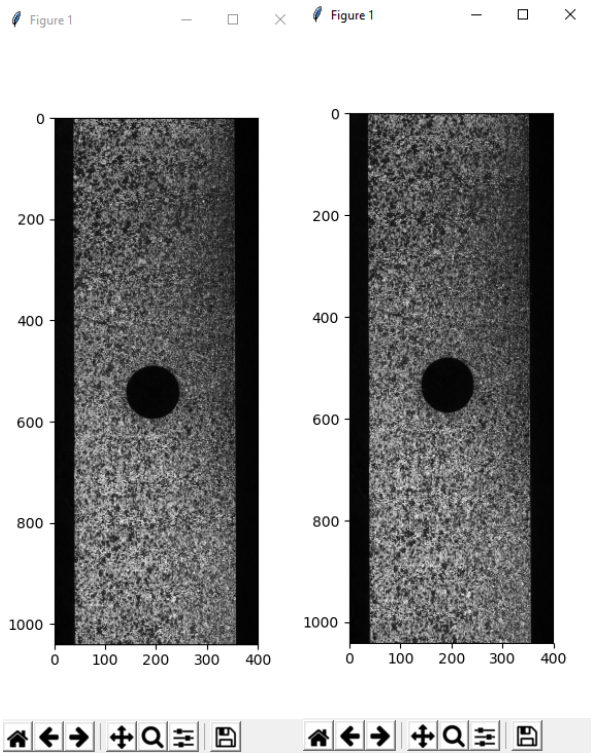


Figure 7.3 – Reference and current images displayed after being entered.

If the cursor is placed over the images, the coordinates of the corresponding pixel can be read, and thus a proper position of the seed can be chosen.

The next parameters to enter are the seed centre, the subset size, the subset spacing, the size of the initial search window, the maximum error allowed, the maximum number of iterations per subset allowed, the half padding, the size of the strain window and the filter image that contains the ROI.

Figure 7.4 – DIC parameters.

After entering the filter image, the valid areas of the original image are shown. It is highly recommended to leave an adequate margin for two reasons: the first one, because the points near the boundaries may disappear in the second image; the second one, because during intermediate iterations, the deformation vector can generate a subset that goes beyond the defined limits.

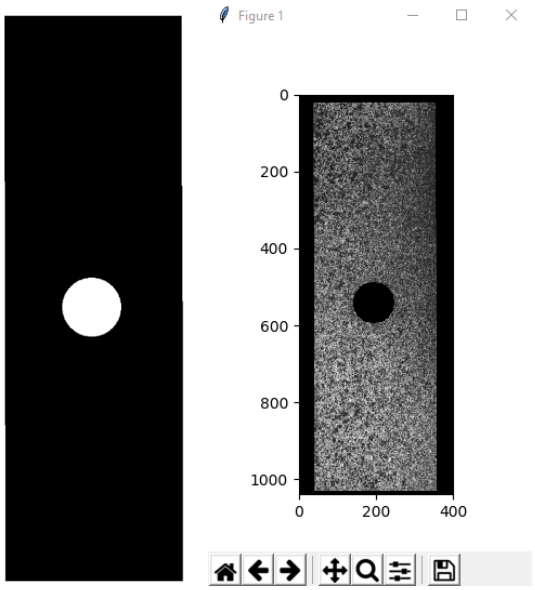


Figure 7.5 – On the left, the loaded filter image. On the right, the filtered image displayed by the program.

After closing the filter image, the program starts calculating the precomputation data, that is, the initial guess and the interpolation array and its derivatives. To check that the program has found a valid initial guess, it displays the NCC matrix. An indication that this result is correct, is to see a point whiter than the rest, having ideally a value of 1. The program also indicates an estimate of the number of valid subsets to be calculated and displays a bar showing the progress.

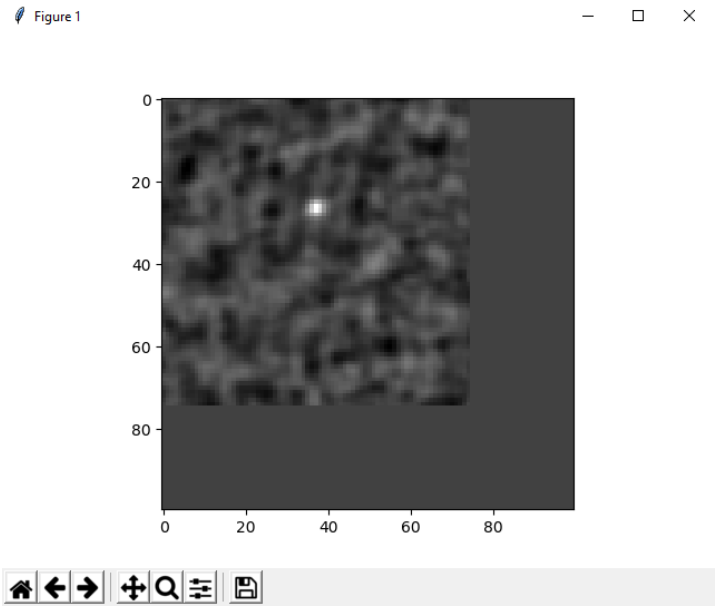


Figure 7.6 – NCC matrix displayed by the program. It is clear that the initial guess has been found correctly.

When the program finishes calculating all the subsets, it displays a colour image showing the path followed in the RG scheme, going from red to blue (following the RGB scale). If there are bad points within the ROI, they will surely appear in blue, but a point appearing in blue does not mean that it is bad, just that it has a lower C_{ZNCC} coefficient. Moreover, the program indicates the time spent so far.

Finally, the user is requested to enter the conversion between pixels and real units, generally millimetres, and the colormap with which the plots will be displayed.

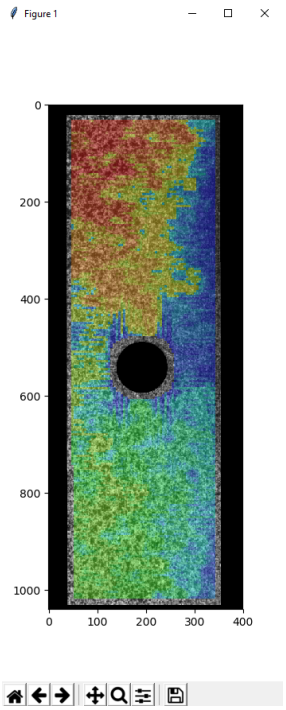


Figure 7.7 – A generated imaged showing the path followed in the RG method.

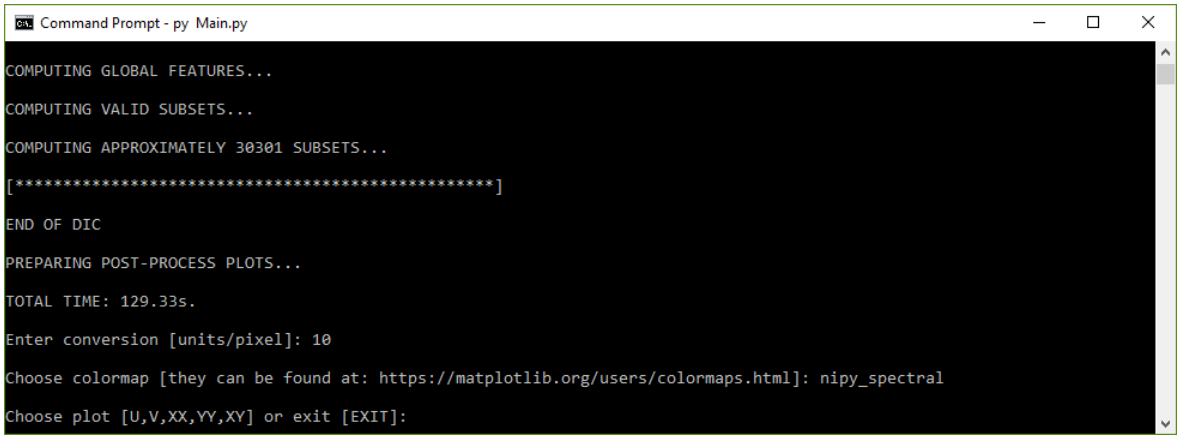


Figure 7.8 – Final steps.

The average calculation time per subset using the set parameters is $4.27\ ms$ (this test has been carried out with an Intel Core i7-7700k and 16 GB of RAM). For larger images, such as 5000×2500 pixels, the peak of the RAM used can reach up to 14 GB, being this the main drawback of the precomputation process, although it is clearly worth the gained speed with respect to computing the coefficients and performing the interpolation individually in each iteration.

The most recommended colormaps are *jet* and *nipy_spectral*. These are the results that can be finally displayed.

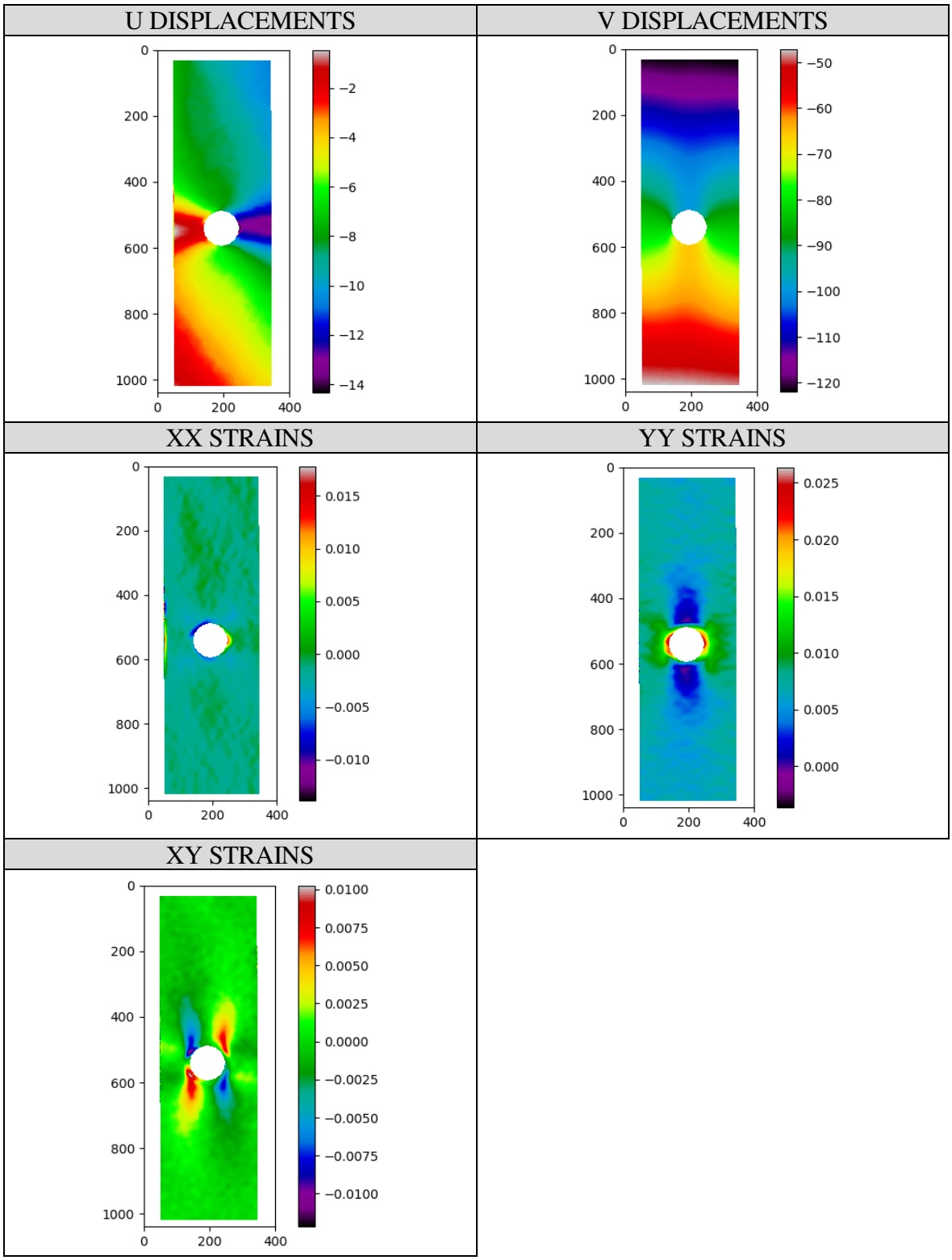


Figure 7.9 – Post-processing plots.

8. Program verification

In order to verify the reliability of the program, a set of different tests have been carried out. The one described in section 8.1 is to check if the program is able to find an appropriate correlation between two images with relative displacements, without yielding an excessive amount of deformation. In section 8.2, additionally, finding the value of the maximum rotation with which the algorithm is capable of locating the seed in the current configuration is intended. Finally, in section 8.3 images obtained from a real experiment will be analysed, and the results will be compared to those obtained using the commercial software GOM Correlate.

8.1. Rigid body translation

The tested reference image, of dimensions $(1040,1392)$, can be downloaded from [2]. It has been proven to have a proper texture, therefore errors that may arise cannot be caused by this factor. The current image has been generated by moving the reference one, arbitrarily, 20 pixels right and 70 pixels down.

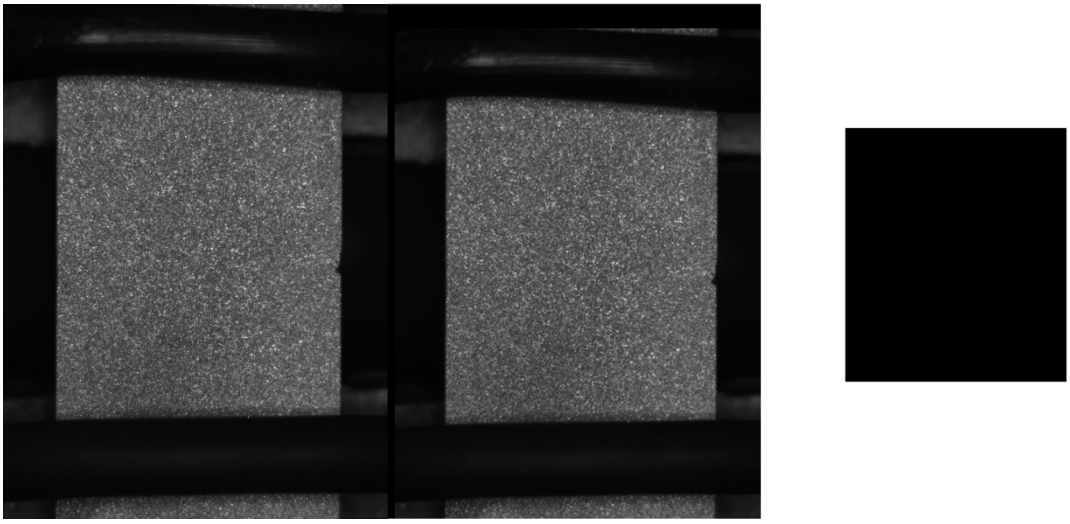
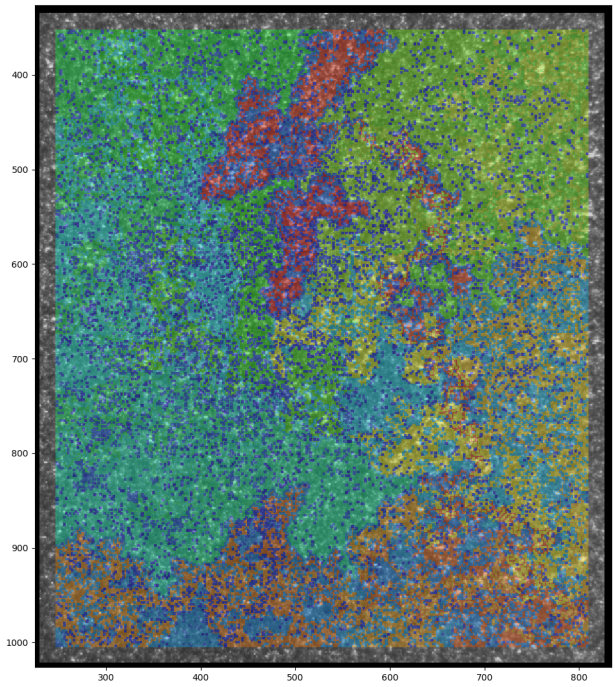


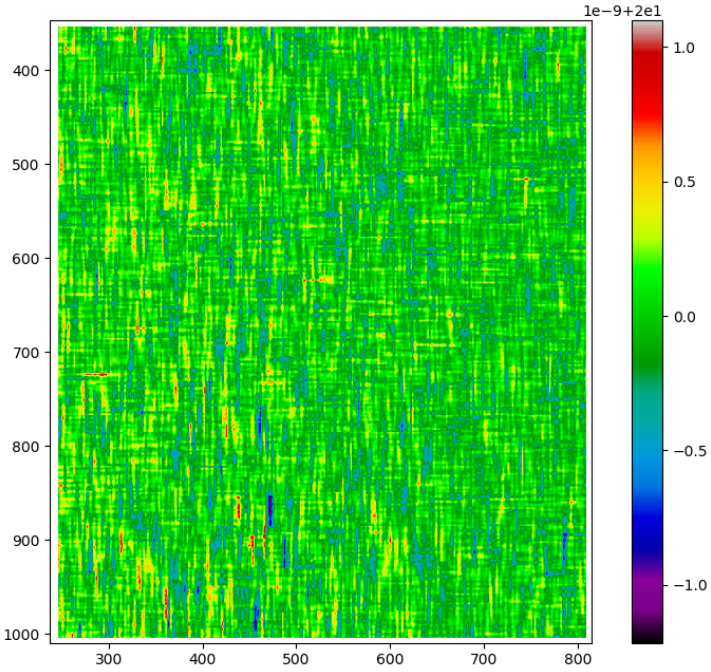
Figure 8.1 – From left to right, reference image, current image and ROI definition.

Given that the deformation is uniform (0 in this case), a large subset size can be chosen. The set parameters are the following.

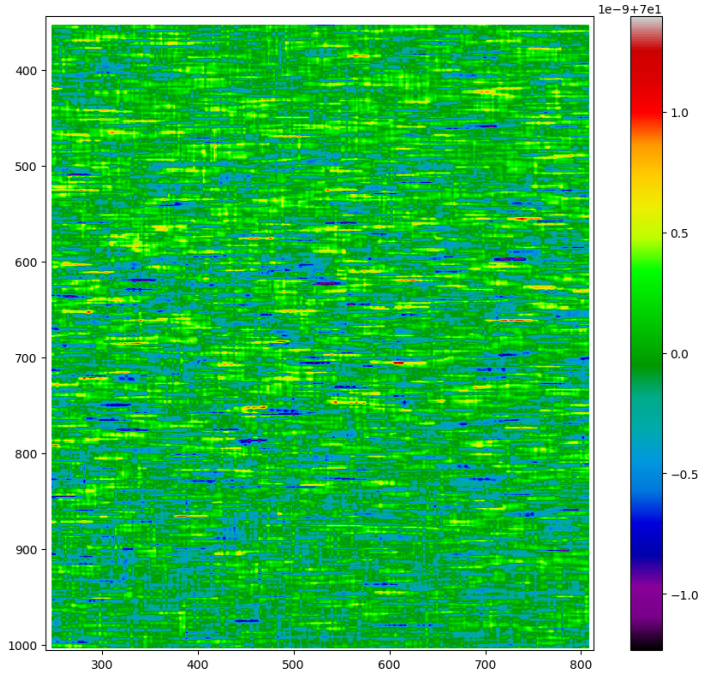
- Degree of interpolation: 5.
- Seed centre: (500,600).
- Subset size: 35.
- Subset spacing: 2.
- Initial search window: 200.
- Allowed error: 10^{-9}.
- Maximum number of iterations per subset: 30.
- Half padding: 3.
- Strain window: 21.
- Units/pixel: 1.

RG PATH


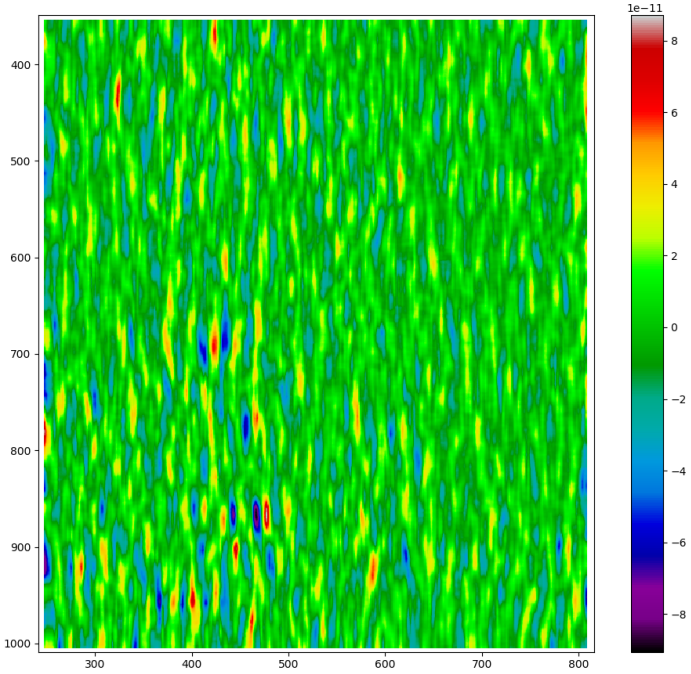
U DISPLACEMENTS



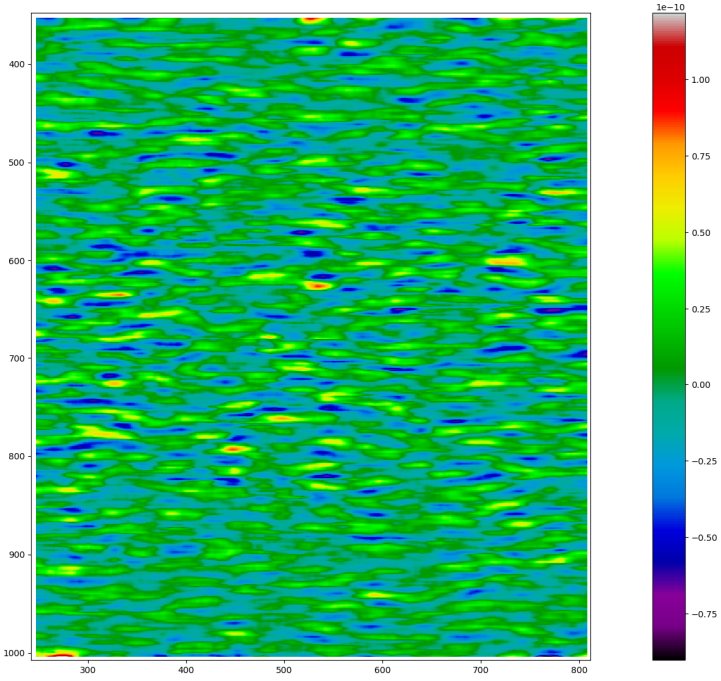
V DISPLACEMENTS



XX STRAINS



YY STRAINS



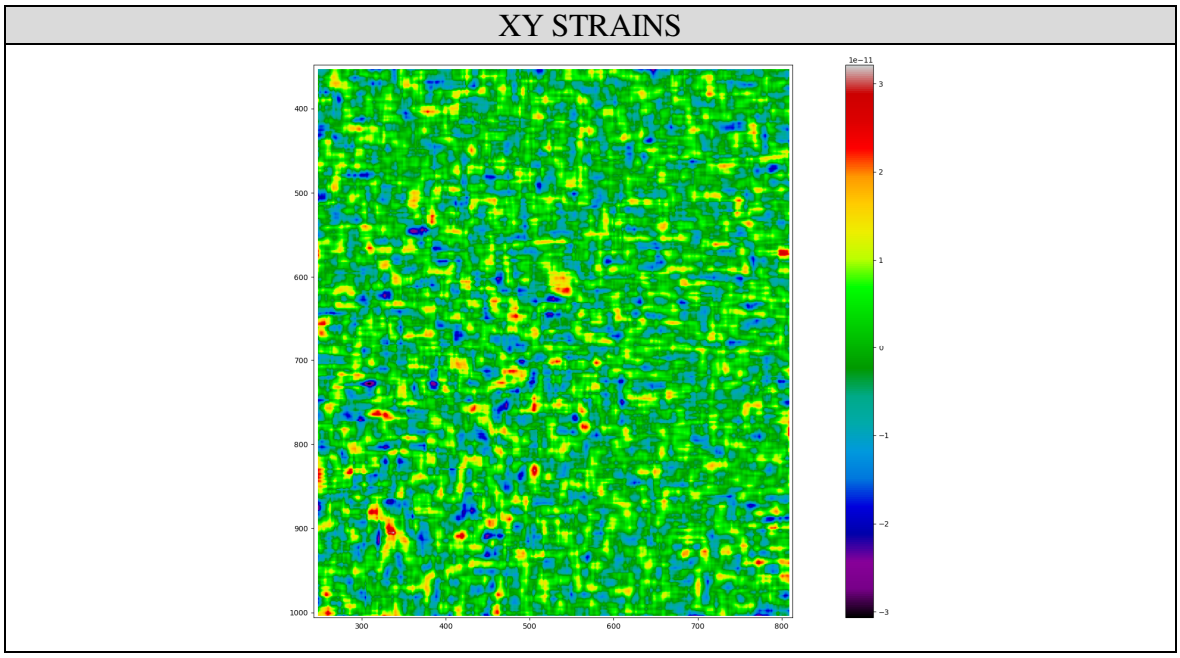


Figure 8.2 – Rigid body translation results.

The fact that the path followed by the RG scheme has some discrete blue points is because everywhere in the image $C_{ZNCC} \approx 1$, so the choice of the next subset is made based on differences around machine precision magnitude. The displacement results are almost exact, $(u, v) = (20{,}70)\ px$: the colour scale goes approximately from -1 to 1, which is multiplied by 10^{-9} and added 20 or 70, as indicates the number above. Therefore, the precision is about 10^{-9}. As for the strains, it is clear that the precision is about 10^{-10}, knowing that the real deformation is 0.

8.2. Rigid body rotation

Several tests have been carried out to find the maximum rotation that still yields appropriate results, being around 15^{o}. It is worth mentioning that this value depends mainly on the ability to find a valid initial guess, as long as there is not any discontinuity in the current configuration, so the results obtained for a particular image just suggest a local estimate. As it has been already mentioned, supposing that the object undergoes a large rotation at some point, but at another point the rotation is small enough, choosing this last point as the seed will yield overall good results, as long as the strains are continuous, because the module of deformation vectors will be increasing between subsets.

The problem of rotating the image using an editor is that the resulting image will not represent the desired rotation accurately, due to the square geometry of pixels. For this reason, the results in this section are those of a computer generated image (using B-spline interpolation) specially designed to test Ncorr. The reference and current images are two MATLAB matrices which can be converted into image files using the function `imwrite`. They can be downloaded from [2].

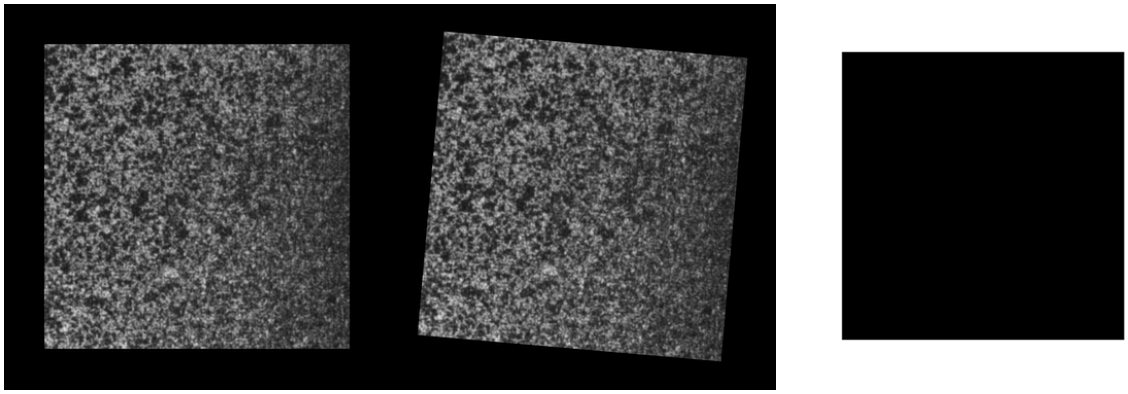
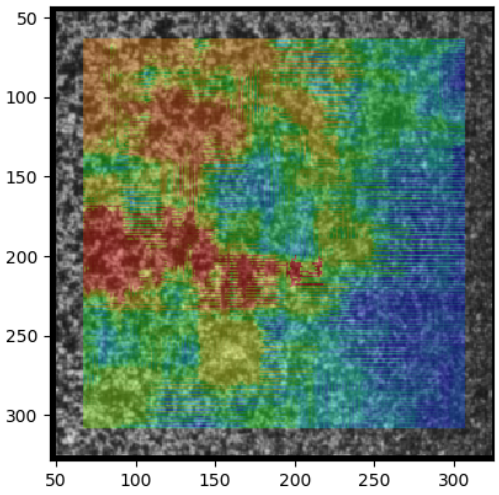


Figure 8.3 - From left to right, reference image, current image (rotated 5º) and ROI definition.

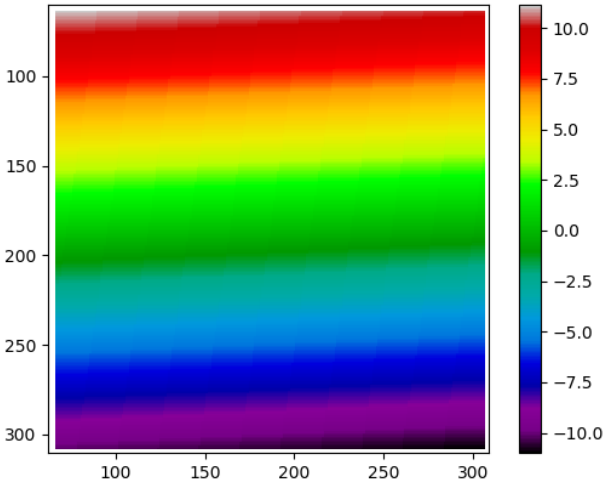
The parameters set for this test are the following:

- Degree of interpolation: 5.
- Seed centre: (200,200).
- Subset size: 35.
- Subset spacing: 1.
- Initial search window: 100.
- Allowed error: 10^{-9}.
- Maximum number of iterations per subset: 30.
- Half padding: 3.
- Strain window: 29.
- Units/pixel: 1.

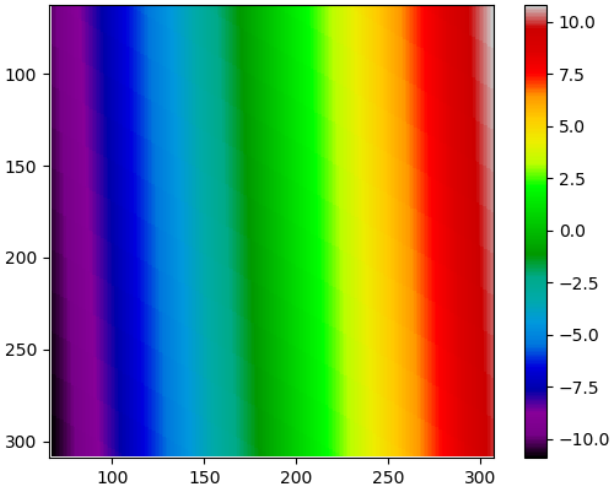
RG PATH



U DISPLACEMENTS



V DISPLACEMENTS



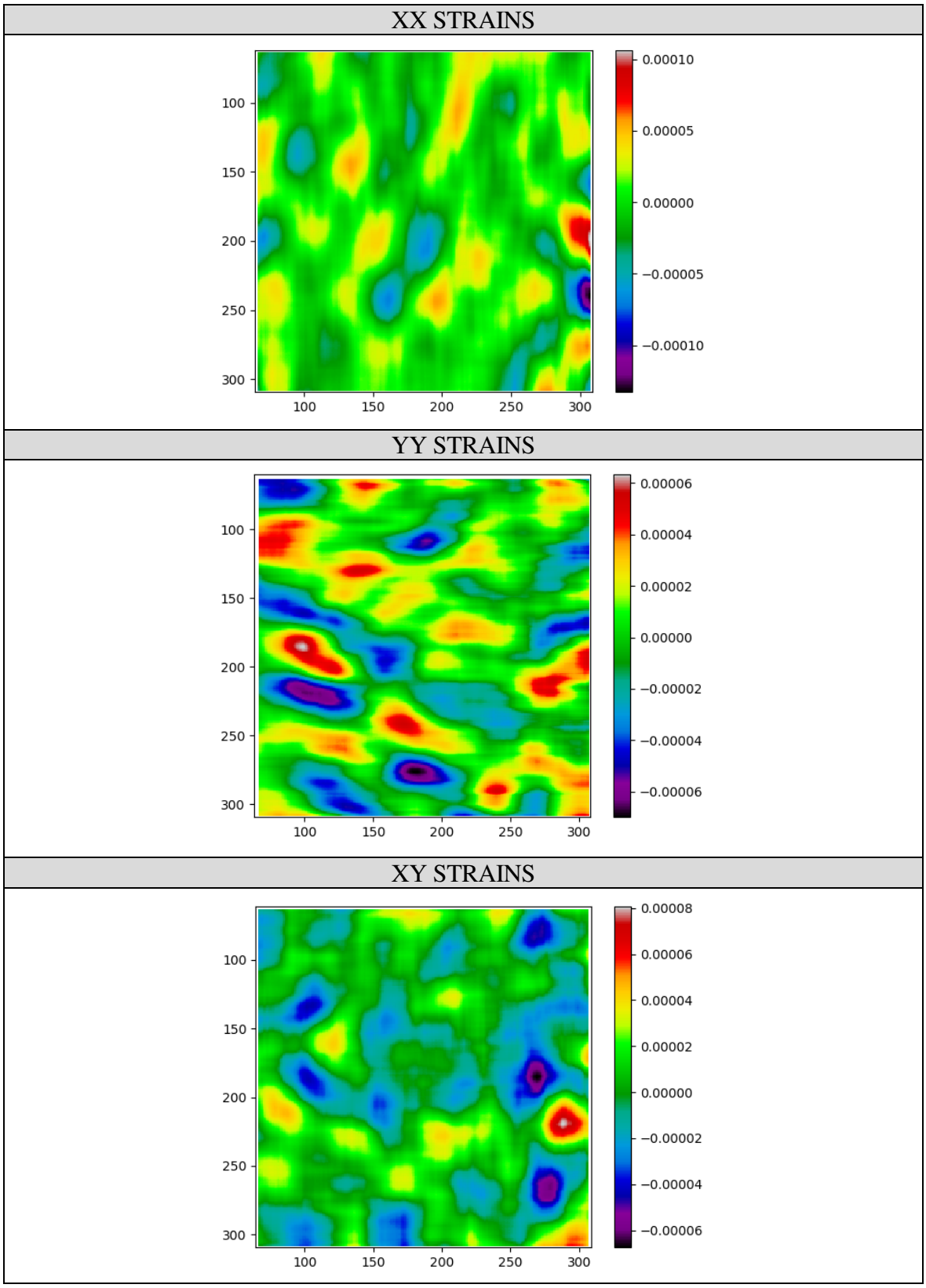


Figure 8.4 – Rigid body rotation results.

The precision is about $5 \cdot 10^{-5}$ (discarding the points near the boundaries), 5 times lower than the Ncorr precision. Given the appearance of the strain plots, it seems that either further data manipulation should be performed (with the results or with the properties of the input images) or some part of the code ought to be checked regarding accuracy, most likely the least squares fitting algorithm. On other hand, it is worth noting that this image was specially generated to test Ncorr accuracy, thus further experiments using real images to test the behaviour in front of rotations are required.

8.3. Slab N-12-3600-3

The last test has been carried out with images taken from a real experiment of the slab N-12-3600-3 under an applied force of 160 kN. The slab has two different parts, the concrete top and the steel-concrete base. Given the discontinuity between the two, two analyses have been carried out, because the program only accepts one seed, and the ROI must have two different areas in order to avoid analysing the discontinuity.

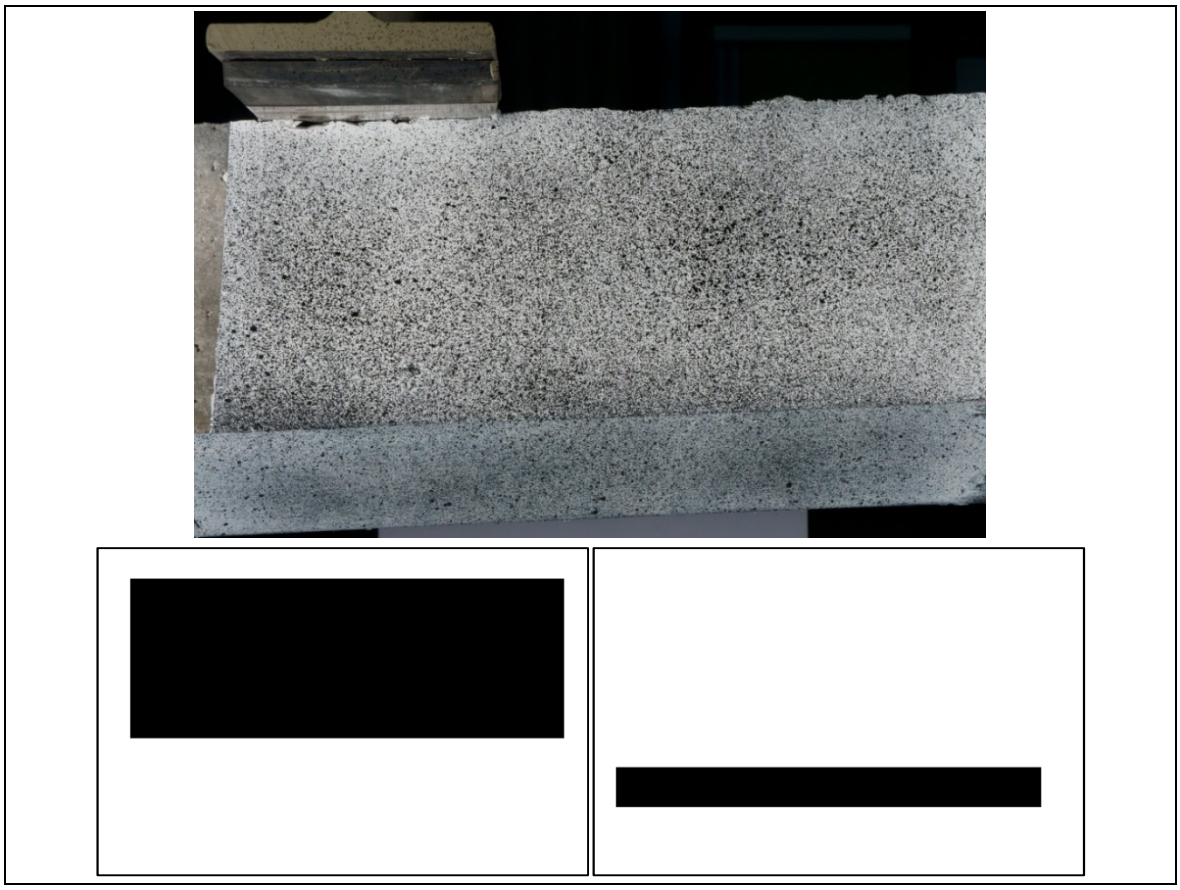
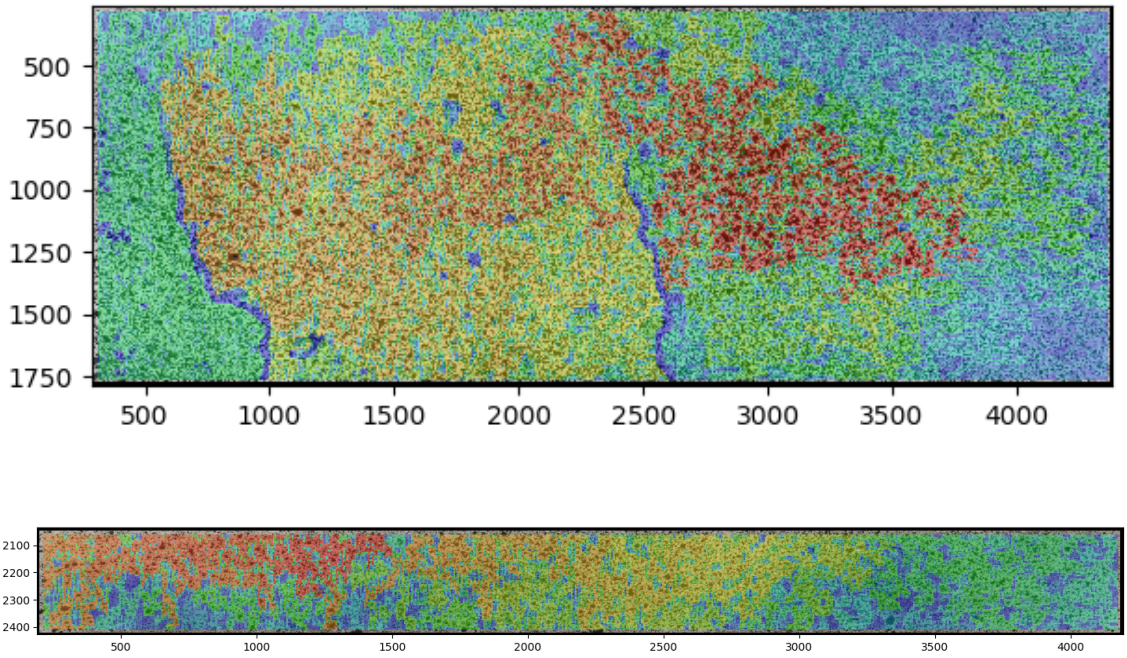


Figure 8.5 – From top to bottom and left to right: reference image, current image, ROI for the first analysis and ROI for the second analysis.

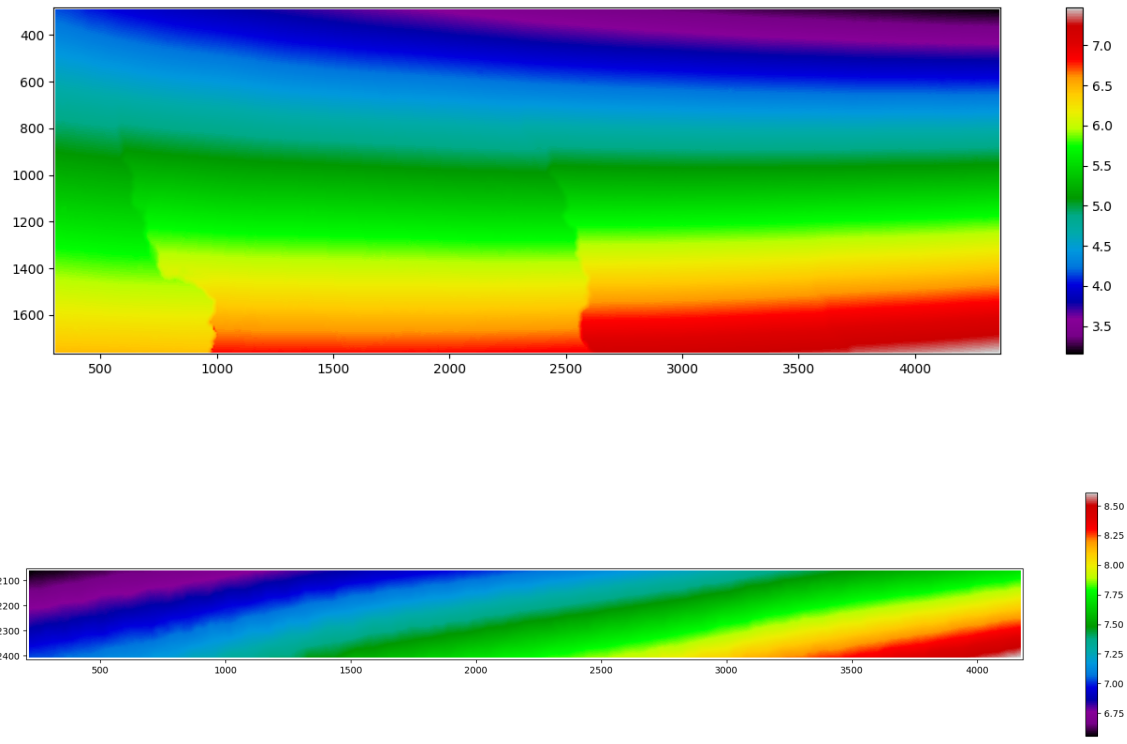
The parameters set for this test are the following:

- Degree of interpolation: 5.
- Seed centre (first analysis): $(3800,1250)$.
- Seed centre (second analysis): $(1400,2100)$.
- Subset size: 25.
- Subset spacing: 7.
- Initial search window: 1000.
- Allowed error: 10^{-9}.
- Maximum number of iterations per subset: 50.
- Half padding: 3.
- Strain window: 23.
- Units/pixel: $0.07282\ mm/px$.

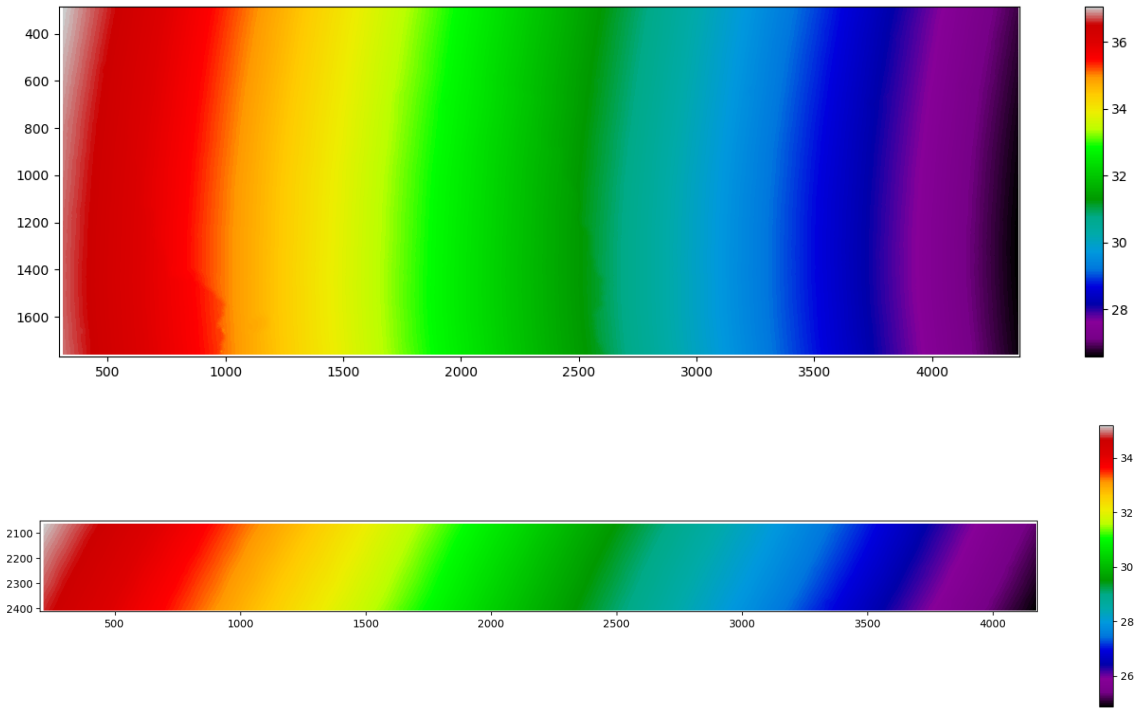
RG PATH



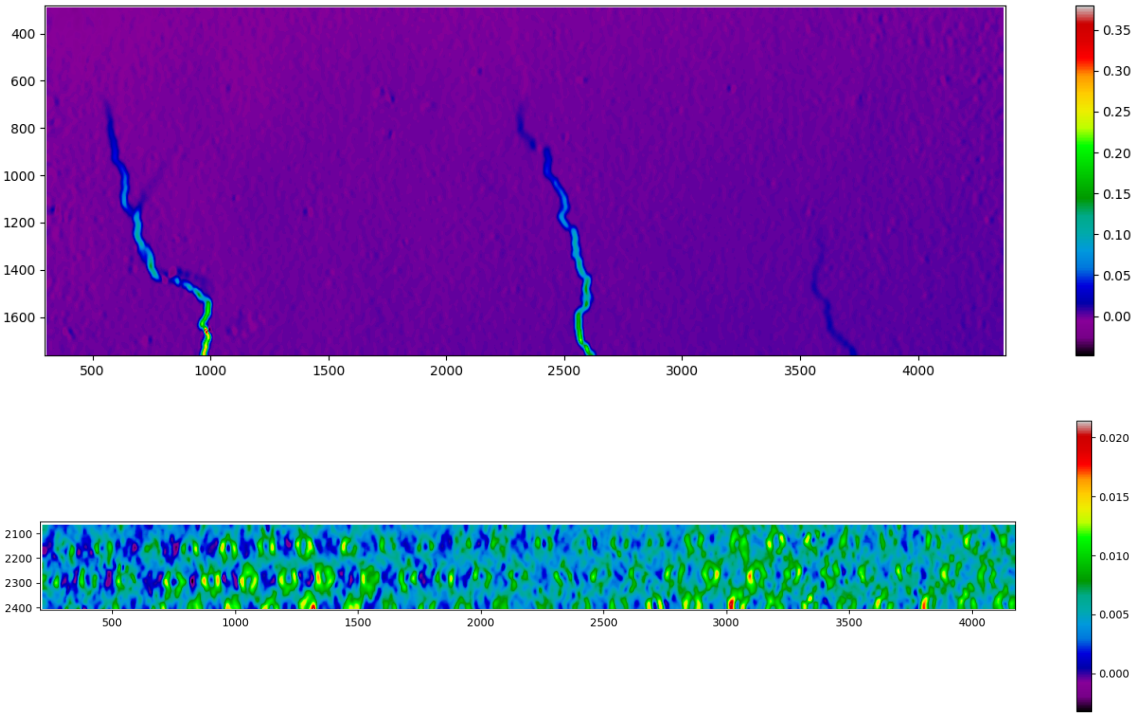
U DISPLACEMENTS



V DISPLACEMENTS



XX STRAINS



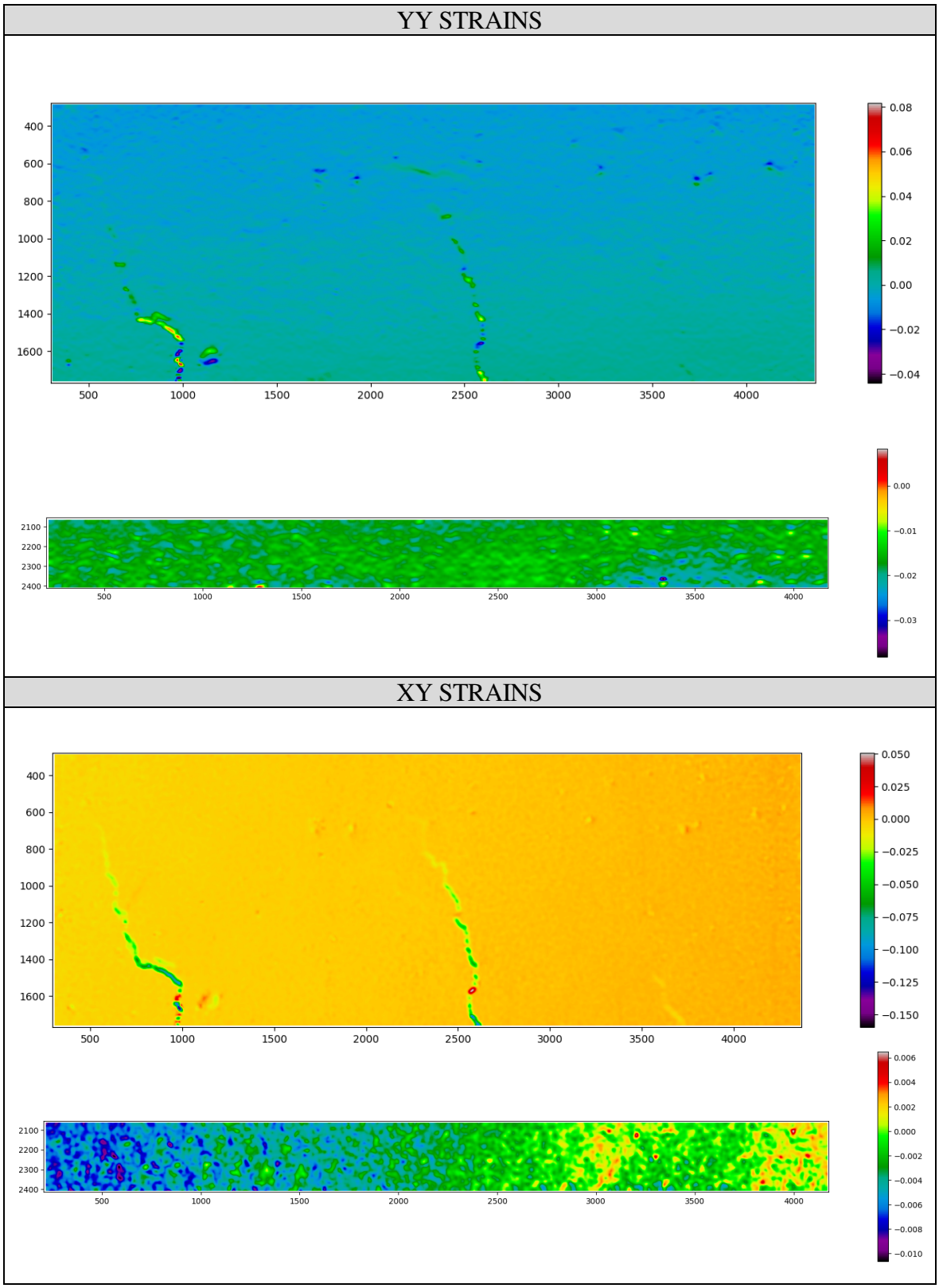


Figure 8.6 – Slab results.

The following figure represents the results obtained using GOM Correlate, for XX strains. The geometry of the cracks that are about to be formed can be clearly seen in both plots, as well as the crown shapes on the steel area. The scale of the plot obtained with GOM Correlate has been applied a 2 $sigma$ scaling factor, so that extreme values do not distort the overall representation. As for the values of the steel area, both plots show that the XX Green strains are approximately between -0.003 and 0.016 (GOM Correlate represents the results as a percentage).

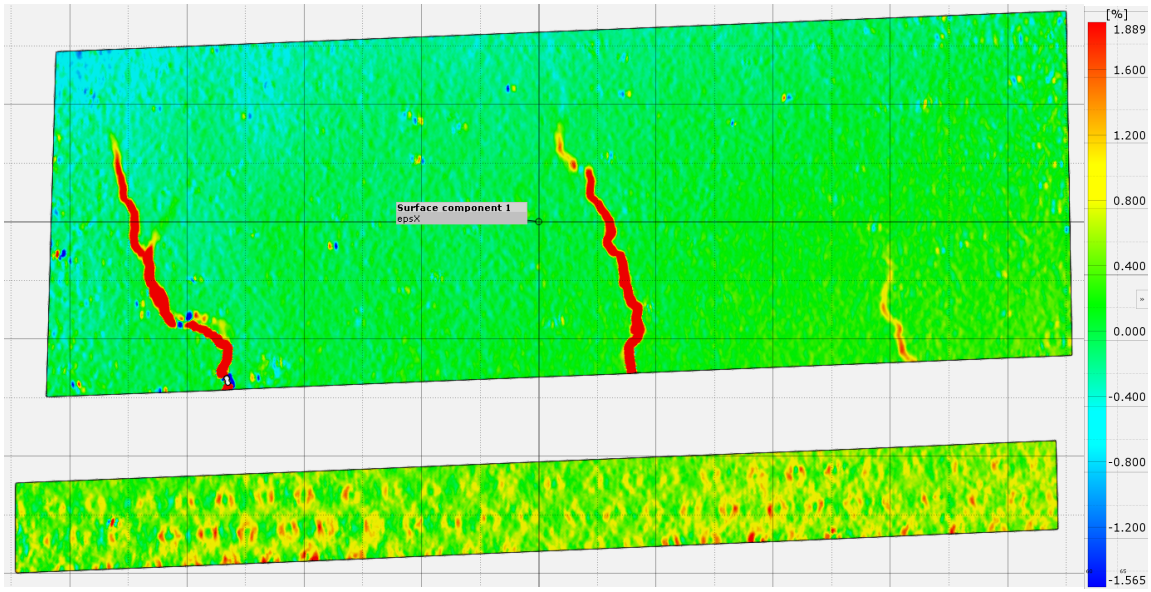
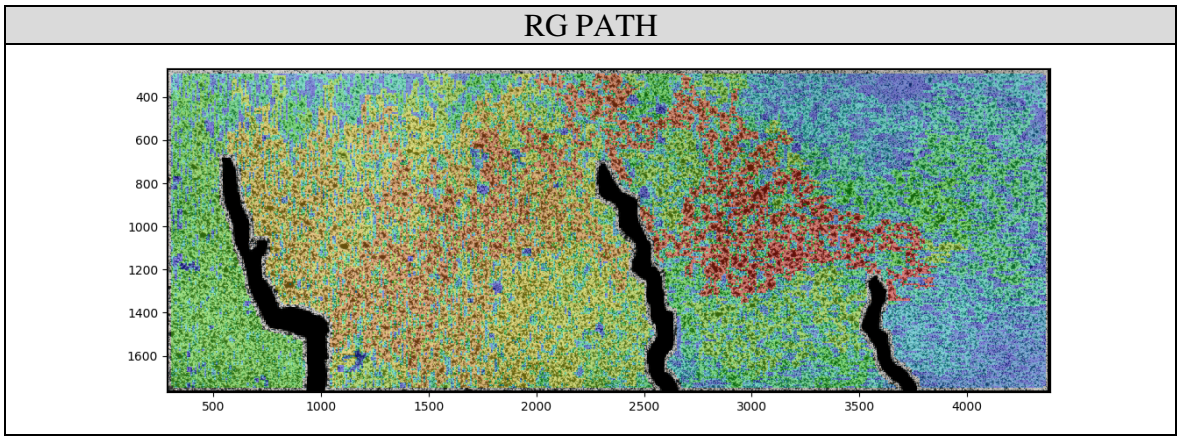
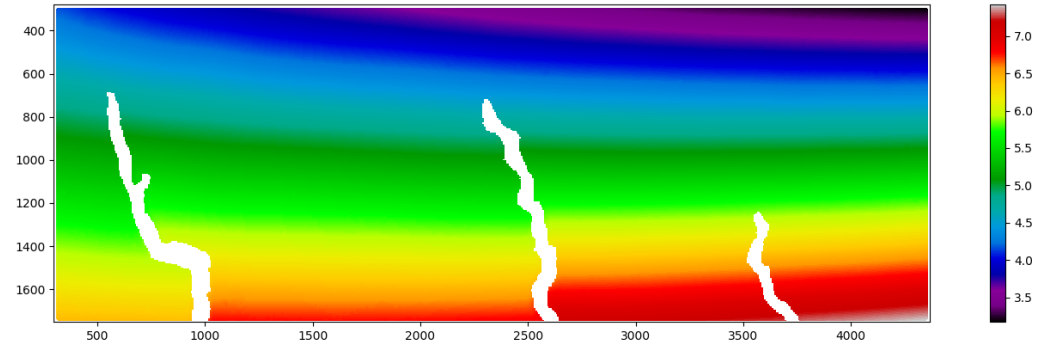


Figure 8.7 – XX Green strains obtained using GOM Correlate.

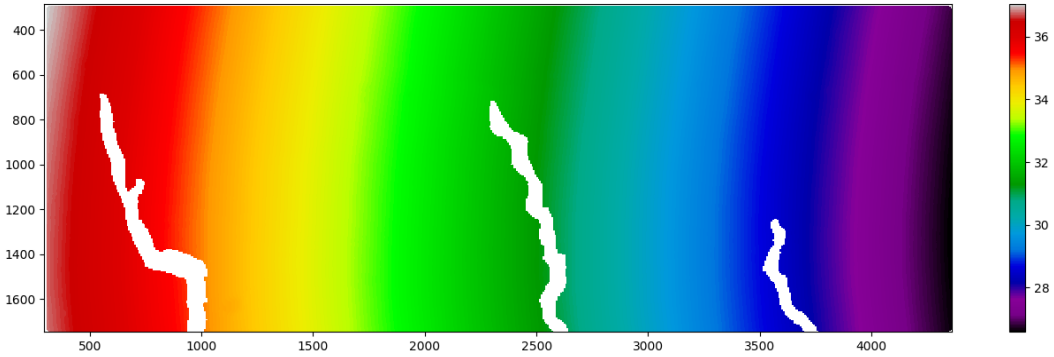
In order to see the results outside the cracks better, another analysis is carried out removing them from the ROI.



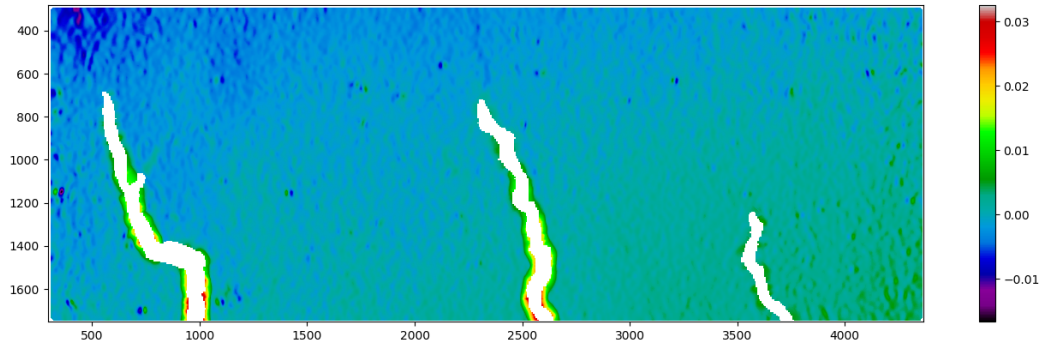
U DISPLACEMENTS



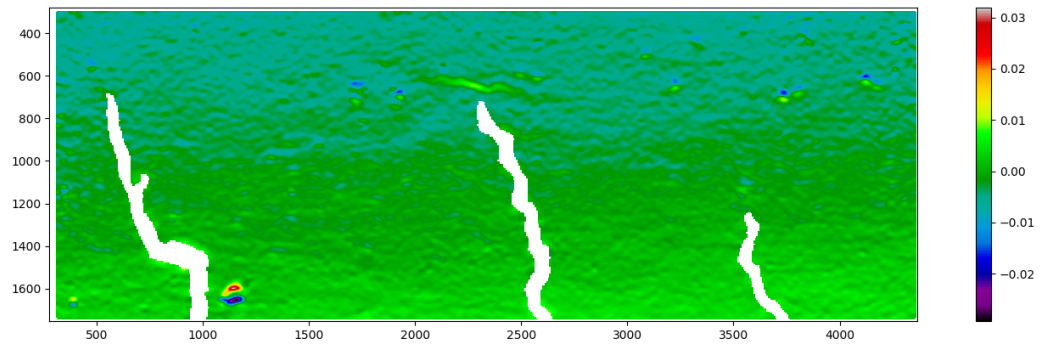
V DISPLACEMENTS



XX STRAINS



YY STRAINS



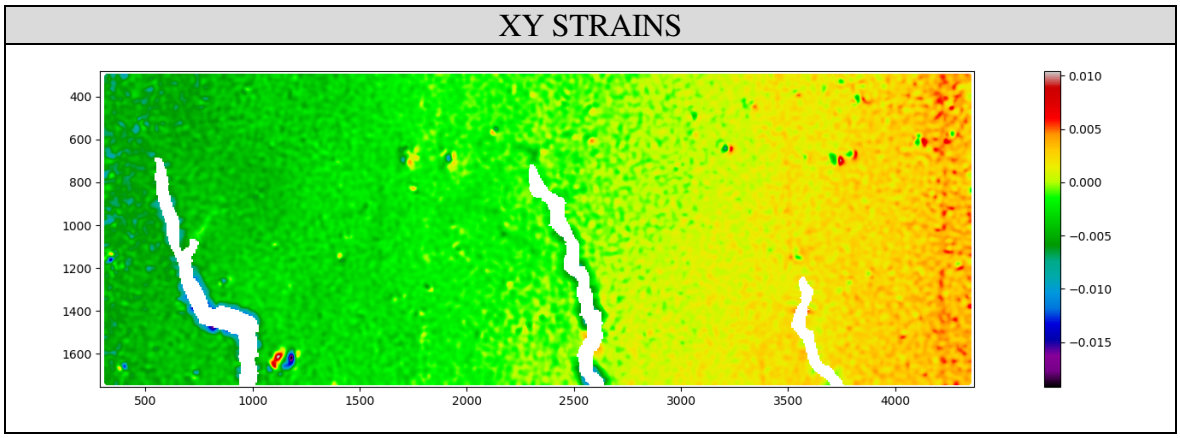


Figure 8.8 – Slab results, discarding the cracks.

The problem of critical points distorting the scale keeps appearing (this time due to the points near the boundaries, and also due to surface imperfections), but the strains outside the cracks can be better observed. Again, comparing the plot with that of GOM Correlate for XX strains, it can be concluded that both give similar results: from top-left to bottom-right, the values go from approximately $[-0.009, -0.006]$ to $[0.002, 0.007]$. Furthermore, the singular points due to surface imperfections are observed in the same positions.

Final conclusions and next steps

It has been proven that the developed application yields overall good results, comparable to those obtained using the commercial package GOM Correlate. In addition, the vectorization of the various operations that the program performs have allowed to increase the speed of computations remarkably, despite being single-threaded. On the other hand, two weak points have been found. The first one is that the post-processing plots have limited features, causing that the representation of the results is not effective when critical points due to proximity to the boundaries or due to surface imperfections are obtained. The second one is the critical RAM peak appearing during the analysis, mainly due to the multidimensional array operations, although `precomp_RAM` may become a solution.

It is clear that further tests need to be carried out to study how varying the DIC parameters affect the results, and also to verify the accuracy of the plane fitting of data, given that the noise obtained in the rigid body rotation test is quite high. It is worth mentioning that this test should be performed taking real images as well, since the used image was generated specifically to test Ncorr, and the B-spline algorithm developed in this project might differ from the one implemented in Ncorr.

Overall, the theoretical framework of a set of modern DIC techniques has been described, and the different parts have been implemented to create a functional application. The following list aims to open a path to improvements that could be tested and implemented in case this project is to be continued.

- Scale-Invariant Feature Transform (SIFT) based algorithm [18]: this algorithm presents a set of advantages over the NCC method if large deformations occur, given that it can already provide the deformation parameters in the initial guess search.

- Post-processing plots: the plots that the program is capable of making at the moment are quite basic, and are very affected by extreme values, which distort the scale. Besides offering more possibilities of representation, such as different strain descriptions, it could also provide a set of tools to exclude areas from the plot, once the analysis is performed (in order to avoid boundary issues).

- Graphic interface: currently, the program is executed via console, and it does not allow to change the parameters without restarting the analysis. A graphic interface would improve the user experience, along with making any changes faster. On the other hand, it could also offer a set of tools to define the ROI directly over the reference image, and to select those areas which need to be analysed more accurately.

- Free subsets: if a particular form of deformation is anticipated, subsets could be defined in such a way that the overall noise is reduced, that is, that they were larger in the direction of more uniform strains, and smaller in the direction of more change. They could also be defined differently according to a certain area of the reference image, or even be described by a function.

- Alternative interpolation method: it has been proven that using B-spline interpolation, along with the IC-GN algorithm, has allowed to construct a precomputed array that can be used anywhere regardless of the current subset coordinates. Nonetheless, this array and the two arrays of derivatives are very large and, despite being calculated by means of vectorization, are still computationally expensive. Some other interpolation methods could be tested to see whether B-spline interpolation offers better results worth the efforts.

- Noise analysis: a feature of the program that could greatly benefit the right choice of the parameters is an estimate of the noise of the image, translating this information to appropriate values for the subset size and the strain window.

- Numba and Cython: these libraries offer a set of tools to run Python code compiling it to native machine instructions, similar in performance to C/C++, after applying some modifications to the former code. Along with vectorization techniques, which have been already implemented, the speed of the program could be greatly improved.

- Multithreading: the RG method must be done serially, thus working with just one processor core. Splitting the image into smaller areas and choosing the corresponding number of seeds could become multithreaded, improving the overall speed of the program. Moreover, these iterations would still need to use Numba or Cython.

Acknowledgments

First of all, I would like to thank Miquel Ferrer for the support and knowledge that has given me throughout these months, as well as all the suggestions and ideas that has proposed for the continuation of the project. Moreover, I would also like to acknowledge the contributions and recommendations made by Frederic Marimon regarding DIC analysis.

Secondly, I would like to mention the great work of Justin Blaber, without which it would have been much more difficult to relate the different parts of the DIC algorithm. Also, I really appreciate the contribution made by Damien André, regarding a small Python-based DIC application which we discussed some programming aspects about.

Finally, I would like to highlight the endless interest and patience of my family.

Bibliography

Bibliographic references

[1] BEHNEL, S., BRADSHAW, R., STEIN, W., FURNISH, G., SELJEBOTN, D., EWING, G., GELLNER, G. *Cython 0.29a0 documentation* [online]. Python Community, 2018 [Date of reference: 23 February 2018]. Available at: <https://cython.readthedocs.io/en/latest/>.

[2] BLABER, J., ADAIR, B., ANTONIOU, J. *Ncorr: Open-Source 2D Digital Image Correlation Matlab Software.* Bethel: Society for Experimental Mechanics, 2015 [Date of reference: 29 December 2017]. Available at: <http://www.ncorr.com/index.php>.

[3] BLABER, J. *Ncorr* [online]. Atlanta: Georgia Institute of Technology, 2018 [Date of reference: 29 December 2017]. DOI: 10.1007/s11340-015-0009-1.

[4] EBERLY, D. *Least Squares Fitting of Data* [online]. Redmond: Geometric Tools, 12 December 2017 [Date of reference: 5 February 2018]. Available at: <https://www.geometrictools.com/Documentation/LeastSquaresFitting.pdf>.

[5] HUNTER, J., DALE, D., FIRING, E., DROETTBOOM, M. *Matplotlib: Release 2.0.2* [online]. Python Community, 10 May 2017 [Date of reference: 22 December 2017]. Available at: <https://matplotlib.org/2.0.2/Matplotlib.pdf>.

[6] LEWIS, J. P. *Fast Normalized Cross-Correlation.* San Francisco: Industrial Light & Magic, 1995. DOI: 10.1016/j.optlaseng.2008.10.014.

[7] LU, H., CARY, P. D. *Deformation Measurements by Digital Image Correlation: Implementation of a Second-order Displacement Gradient*. Berlin: Experimental Mechanics, December 2000. DOI: 10.1007/BF02326485.

[8] MÂŢIU-IOVAN, L., FRIGURĂ-ILIASA, F. M., VĂTĂU, D. *A method to determine the coefficients in B-spline interpolation.* Timisoara: IIEE, 26 December 2007. DOI: 10.1109/EURCON.2007.4400395.

[9] MATTHEWS, I., BAKER, S. *Lucas-Kanade 20 Years On: A Unifying Framework.* Pittsburgh: Kluwer Academic Publishers, 7 February 2003. DOI: 10.1023/B:VISI.0000011205.11775.fd.

[10] MORDVINTSEV, A., K, A. *OpenCV-Python Tutorials Documentation: Release 1* [online]. Python Community, 5 November 2017 [Date of reference: 22 December 2017]. Available at: <https://readthedocs.org/projects/opencv-python-tutroals/downloads/pdf/latest/>.

[11] NUMPY COMMUNITY. *NumPy Reference: Release 1.13.0* [online]. Python Community, 10 June 2017 [Date of reference: 22 December 2017]. Available at: <https://docs.scipy.org/doc/numpy-1.13.0/numpy-ref-1.13.0.pdf>.

[12] PAN, B. *Reliability-guided digital image correlation for image deformation measurement.* Washington D. C.: Optical Society of America, 4 March 2009. DOI: 10.1364/AO.48.001535.

[13] PAN, B., ASUNDI, A., XIE, H., GAO, J. *Digital image correlation using iterative least squares and pointwise least squares for displacement field and strain field measurements.* Amsterdam: Elsevier, 20 February 2009. DOI: 10.1.1.21.6062.

[14] PAN, B., XIE, H., WANG, Z. *Equivalence of digital image correlation criteria for pattern matching.* Washington D. C.: Optical Society of America, 30 September 2010. DOI: 10.1364/AO.49.005501.

[15] SCIPY COMMUNITY. *SciPy Reference Guide: Release 0.19.1* [online]. Python Community, 21 June 2017 [Date of reference: 22 December 2017]. Available at: <https://docs.scipy.org/doc/scipy-0.19.1/scipy-ref-0.19.1.pdf>.

[16] THÉVENAZ, P. *Interpolation Revisited.* Lausanne: IEEE, July 2000. DOI: 10.1109/42.875199.

[17] UNSER, M. *Splines: A Perfect Fit for Signal and Image Processing*. Lausanne: IEEE, November 1999. DOI: 10.1109/79.799930.

Additional bibliography

[18] PAN, B., DAFANG, W., YONG, X. *Incremental calculation for large deformation measurement using reliability-guided digital image correlation.* Amsterdam: Elsevier, April 2012. DOI: 10.1016/j.optlaseng.2011.05.005.